

USENIX

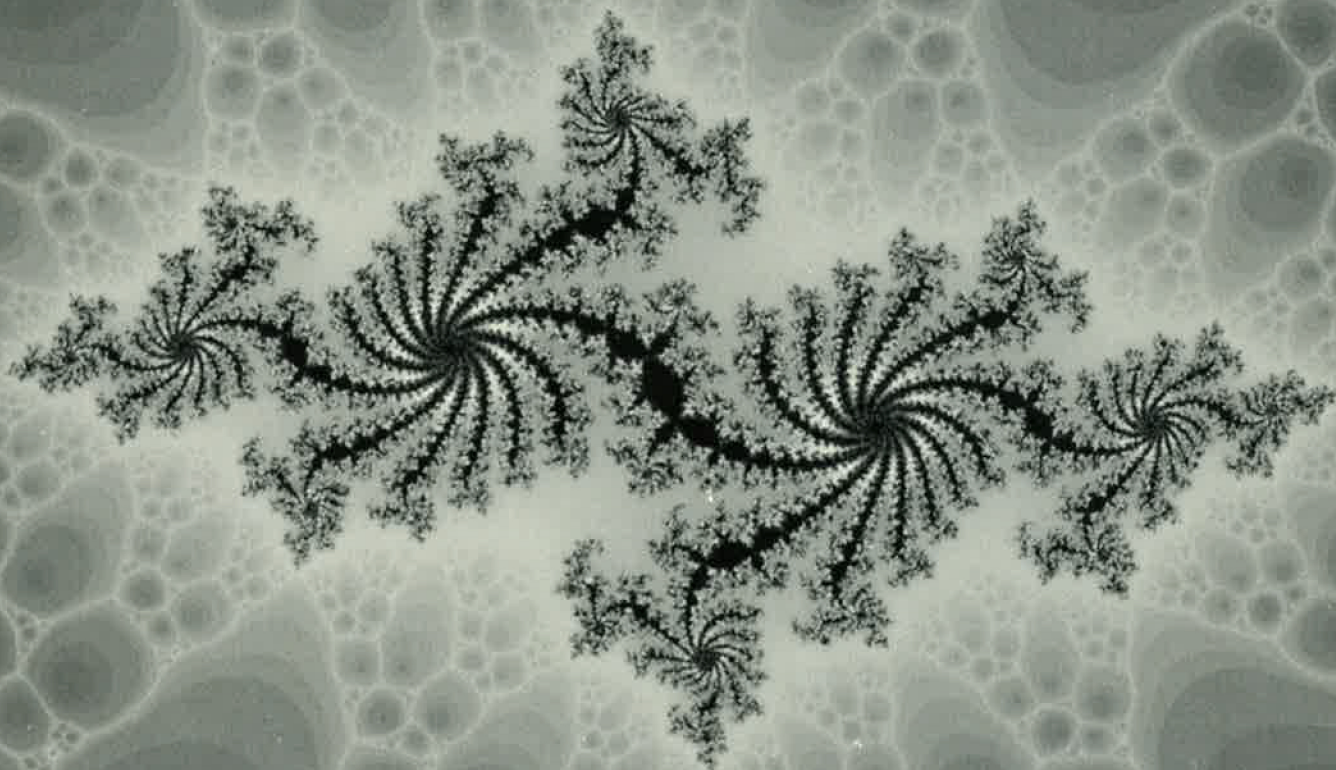
CINCINNATI CONFERENCE PROCEEDINGS

USENIX SUMMER 1993

◆ ◆ ◆ **TECHNICAL CONFERENCE** ◆ ◆ ◆

JUNE 21-25, 1993 ◆ CINCINNATI, OHIO

CONFERENCE PROCEEDINGS



**THE UNIX AND ADVANCED COMPUTING
PROFESSIONAL AND TECHNICAL ASSOCIATION**

SUMMER

1993

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$25 for members and \$33 for nonmembers.
Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1993 Winter San Diego	1987 Summer Phoenix
1992 Summer San Antonio	1987 Winter Washington, DC
1992 Winter San Francisco	1986 Summer Atlanta
1991 Summer Nashville	1986 Winter Denver
1991 Winter Dallas	1985 Summer Portland
1990 Summer Anaheim	1985 Winter Dallas
1990 Winter Washington, DC	1984 Summer Salt Lake City
1989 Summer Baltimore	1984 Winter Washington, DC
1989 Winter San Diego	1983 Summer Toronto
1988 Summer San Francisco	1983 Winter San Diego
1988 Winter Dallas	

1993 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-50-2

This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.



USENIX Association

**Proceedings of the
Summer 1993 USENIX Conference**

**June 21-25, 1993
Cincinnati, Ohio, USA**

TABLE OF CONTENTS

Wednesday (9:00 - 10:20)

Keynote Speaker: *Bruce Tognazzini, SunSoft, Inc*

Wednesday (11:00 - 12:30)

Session Chair: **M. Kirk McKusick**

Call Path Profiling of Monotonic Program Resources in UNIX.....1
Robert J. Hall, Aaron J. Goldberg, AT&T Bell Laboratories

Computer System Performance Problem Detection Using Time Series Model15
Peter Hoogenboom, Jay Lepreau, Center for Software Science, Department of Computer Science, University of Utah

Design and Implementation of a Simulation Library Using Lightweight Processes33
Janche Sang, Ke-hsiung Chung, Vernon Rego, Department of Computer Sciences, Purdue University

Wednesday (2:00 - 3:30)

Session Chair: **Matt Blaze**

The Restore-o-Mounter: The File Motel Revisited.....45
Joe Moran, Bob Lyon, Legato Systems Incorporated

The Autofs Automounter59
Brent Callaghan, Satinder Singh, SunSoft, Inc.

Discovery and Hot Replacement of Replicated Read-Only File Systems,
with Application to Mobile Computing.....69
Erez Zadok, Dan Duchamp, Computer Science Department, Columbia University

Wednesday (4:00 - 6:00)

Session Chair: **Pat Parseghian**

X Through the Firewall, and Other Application Relay.....87
G. Winfield Treese, MIT Laboratory for Computer Science and Digital Equipment Corporation; Alec Wolman, University of Washington and Digital Equipment Corporation

The Ferret Document Browser.....101
Howard P. Katseff, Thomas B. London, AT&T Bell Laboratories

LADDIS: The Next Generation in NFS File Server Benchmarking111
Mark Wittle, Data General Corporation; Bruce E. Keith, Digital Equipment Corporation

Design and Implementation of a Multimedia Protocol Suite in a BSD UNIX Kernel129
K. Lakshman, Giri Kuthethoor, Raj Yavatkar, Dept. of Computer Sciences, University of Kentucky

Thursday (11:00 - 12:30)

Session Chair: **Steve Kleiman**

The Spring Nucleus: A Microkernel for Objects147
Graham Hamilton, Panos Kougiouris, Sun Microsystems Laboratories Inc.

"Stacking" Vnodes: A Progress Report161
Glenn C. Skinner, Thomas K. Wong, SunSoft, Inc.

Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space.....	175
<i>Curtis Yarvin, Richard Bukowski, Thomas Anderson, Division of Computer Science, University of California, Berkeley</i>	

Thursday (2:00 - 3:30)

Session Chair: Nathaniel Borenstein

Integrating Handwriting Recognition into UNIX	187
<i>James Kempf, Sun Microsystems Computer Corporation</i>	
Optimizing UNIX Resource Scheduling for User Interaction	205
<i>Steve Evans, Bart Smaalders, Dave Singleton, SunSoft, Inc.</i>	
AudioFile: A Network-Transparent System for Distributed Audio Applications	219
<i>James Gettys, Thomas M. Levergood, Andrew C. Payne, Lawrence C. Stewart, Digital Equipment Corporation and G. Winfield Treese, MIT Laboratory for Computer Science and Digital Equipment Corporation</i>	

Friday (9:00 - 10:20)

Session Chair: J. R. Oldroyd

Fast and Flexible Shared Libraries.....	237
<i>Douglas B. Orr, Jay Lepreau, John Bonn, Robert Mecklenburg, Center for Software Science, Department of Computer Science, University of Utah</i>	
High Performance Dynamic Linking Through Caching	253
<i>Michael N. Nelson, Graham Hamilton, Sun Microsystems Laboratories, Inc.</i>	
The Shell as a Service.	267
<i>Glenn Fowler, AT&T Bell Laboratories</i>	

Friday (11:00 - 12:30)

Session Chair: Jeffrey Mogul

A User-Level Replicated File System.....	279
<i>Glenn Fowler, Yennun Huang, David Korn, Herman Rao, AT&T Bell Laboratories</i>	
sfs: A Parallel File System for the CM-5	291
<i>Sue J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, Richard Wheeler, Thinking Machines Corporation</i>	
Adaptive Block Rearrangement Under UNIX	307
<i>Sedat Akyurek, Kenneth Salem, Dept. of Computer Science, University of Maryland</i>	

AUTHOR INDEX

Sedat Akyurek	307	Bob Lyon	45
Thomas Anderson	175	Robert Mecklenburg	237
John Bonn	237	Ewan D. Milne	291
Richard Bukowski	175	Joe Moran	45
Brent Callaghan	59	Andy Nanopoulos	291
Ke-hsiung Chung	33	Michael N. Nelson	253
Dan Duchamp	69	William Nesheim	291
Steve Evans	205	Douglas B. Orr	237
Glenn Fowler	267, 279	Andrew C. Payne	219
James Gettys	219	Herman Rao	279
Aaron J. Goldberg	1	Vernon Rego	33
Robert J. Hall	1	Kenneth Salem	307
Graham Hamilton	147, 253	Janche Sang	33
Peter Hoogenboom	15	Satinder Singh	59
Yennun Huang	279	Dave Singleton	205
Marshall Isman	291	Glenn C. Skinner	161
Howard P. Katseff	101	Bart Smaalders	205
Bruce E. Keith	111	Lawrence C. Stewart	219
James Kempf	187	G. Winfield Treese	87, 219
David Korn	279	Richard Wheeler	291
Panos Kougiouris	147	Mark Wittle	111
Giri Kuthethoor	129	Alec Wolman	87
K. Lakshman	129	Thomas K. Wong	161
Jay Lepreau	15, 237	Curtis Yarvin	175
Thomas M. Levergood	219	Raj Yavatkar	129
Sue J. LoVerso	291	Erez Zadok	69
Thomas B. London	101		

ACKNOWLEDGEMENTS

Program Committee

David Rosenthal, *Program Chair, Sun Microsystems*
Matt Blaze, *AT&T Bell Laboratories*
Nathaniel Borenstein, *Bellcore*
Bob Gray, *U S WEST Advanced Technologies*
Steve Kleiman, *Sun Microsystems*
John Kohl, *Digital Equipment Corporation*
M. Kirk McKusick, *University of California, Berkeley*
Jeffrey Mogul, *Digital Equipment Corporation*
JR Oldroyd, *Instruction Set*
Pat Parseghian, *AT&T Bell Laboratories*
Dennis Ritchie, *AT&T Bell Laboratories*

Readers

Ian Howell
Darren R. Hardy
Bruce Hilyer
Linda Julien
Howard Katseff
Adam Moskowitz
Kent Peacock
Beth Robinson
Peg Schafer
Nick Stoughton
Theodore Ts'o

Proceedings Production

Carolyn S. Carr, *USENIX Association*
Malloy Lithographing

Call Path Profiling of Monotonic Program Resources in UNIX

by

Robert J. Hall

Aaron J. Goldberg

AT&T Bell Laboratories

600 Mountain Ave.

Murray Hill, NJ 07974

Abstract

Practical performance improvement of a complex program must be guided by empirical measurements of its resource usage. Essentially, the programmer wants to know where in the source code the program is inefficient and why this is so. The process interface of UNIX System V (`proc(4)`) provides access to the raw data (e.g. time, faults, traps, and system calls) necessary to answering the why question, but gives no guidance in answering the where question. This paper describes a novel approach to the latter, *Call Path Profiling*, which is both more informative and more closely tied to the process of program optimization than either trace-based or `prof/gprof`-like approaches. In addition, by viewing consumption of a resource as the ticking of a clock, we generalize the interval-based sampling approach of time profilers to arbitrary monotonic resources. The approach is embodied in several prototypes, including `CPPROF` which operates under System V.

1 Introduction

In order to effectively improve the performance of a complex program, it is essential that one understand both *why* the program is too slow or big (i.e., what resources it uses) and *where* it consumes those resources significantly. This allows one to focus time and attention on re-examining critical design decisions, while avoiding insignificant optimizations that needlessly destroy the program's original clear structure. While there has been some progress in automated static analysis of program performance[8, 3], prospects for a practical tool are dim. Thus, in practice one still turns to *profiling*: measuring resource usage empirically[7].

Previous research (see Section 3) has treated the "where" question only crudely. In particular, most profilers (e.g. UNIX's `prof(1)`) report only the costs of executing function bodies. While this is useful in locating inefficiently coded function bodies, it is rare that inefficiency is localized to one routine. Far more common is when a general routine, efficiently coded for a general task, is called in a specialized context, doing more work than necessary. The programmer can optimize by creating a specialized version which is only called in the optimizable context. A profiler should report resource usage in terms of these *optimizable program contexts*, because they are more likely to be directly optimizable.

Call path profiling[6] is a technique for describing "where" in terms of optimizable program contexts. This paper reports on an implementation of call path profiling of monotonic program resources (e.g. time, faults, traps, system calls, etc.; see Section 2) under the process filesystem (`proc(4)`) abstraction[2] of UNIX System V. A first pass collects raw resource data about the target process via `proc(4)`, implementing an "interval timer" for each resource of interest. Specifically, we view consumption of a resource as the ticking of a clock. Each time the resource clock sweeps out a full interval, we pause the target process and record in a *cpprofd* file a *stack sample*: the program counter and the sequence of return addresses stored on the current program call stack. This does not result in excessive overhead primarily because we are sampling (recording infrequent but randomly distributed data) instead of tracing (recording all data); we trade off perfect accuracy for low overhead. The second pass analyzes the stack samples in the

cpprofd file and puts the user in a simple command loop that allows interactive exploration of call path profiles of various resources viewed from different root functions.

While call path profiling has previously been used in Lisp environments for time and space, the `proc(4)` filesystem provides a unique opportunity for profiling a wide variety of interesting resources in a uniform and portable way. In addition, our research points to the need for a “generalized interval timer” service which we believe should be provided by the UNIX kernel for all monotonic system resources in a way similar to the interval timer (`setitimer(2)`).

2 Monotonic Resources and `proc(4)`

People usually complain about performance when programs are too slow. Thus, the typical profiler reports on the consumption of the single resource, execution time. However, in many cases one wishes to know why the program spends time in a given place, and this requires understanding its usage of other resources. For example, a particular call to a floating point routine may take much longer than expected due to floating point exception processing. A time profiler does not help one to understand why the call is slow, but a profile of where the floating point exceptions occur in a program informs the user directly. Similarly, one might wish to profile any or all of the resources listed below.

```
user time
real time
floating point exceptions
page faults
input/output time
bytes read or written
system or function calls
...
```

The key property these all share is that their effects increase *monotonically* over time; that is, the total cost over the program's execution is equal to the sum of the costs incurred by individual subroutines. For example, each page fault adds to the total run-time cost of the program. By contrast, some resources, such as stack space, do not have this property. Just because a given subroutine invocation uses k bytes of stack space does not imply that the whole program execution must use an additional k bytes; it may actually reuse the same k bytes allocated by some previous subroutine invocation. A similar example is parallel execution time. Our approach does not apply directly to these non-monotonic resources, so by “resource” we will always mean “monotonic resource.”

Another key property of the resources listed above is that a monitoring process can acquire raw data about them for another process through the `proc(4)` filesystem¹. For example, the monitoring process can arrange to be notified of every page fault, trap, or system call encountered by the monitored process. Our System V prototype, CPPROF, exploits this functionality.

3 Related Work

To illustrate call path profiling and contrast it with other profilers, it will be useful to refer to the (contrived) example program, `process_db`, shown below. Section 6 will discuss more realistic programs to which we have applied our profiler.

¹To monitor time resources, CPPROF uses the kernel's interval timer facilities (`setitimer(2)`)


```

main()
{
    DB db = read_db();
    print_salary_stats(uniqueify_db(db));
}

DB_RECORD **uniqueify_db(DB db)
{
    DB_RECORD **dbptrs = build_db_ptrs(db);
    qsort(dbptrs, name_field_lt);
    merge_adjacent_records();
    return(dbptrs);
}

print_salary_stats(DB_RECORD **dbptrs)
{
    int *salaries = extract_salaries();
    qsort(salaries, integer_lt);
    stat_summary(salaries);
}

```

Essentially, this program reads a database of employee records, sorts it into order by name field eliminating duplicate records, and then extracts and sorts the salaries. Finally it prints various statistics about them, such as range, median, etc. Note that the programmer realizes that sorting the database directly would be inefficient, because `qsort` would have to swap all the bytes of the records, so `uniqueify_db` creates, manipulates, and returns an array of pointers to database records.

Suppose now that we run this program on a database too large to fit in the physical memory of our machine.

UNIX's `prof(1)` reports the amount of time spent in each function body. For `process_db`, it would report that essentially all the time is spent in `qsort`. While true, this doesn't help the optimizer because `qsort(3)`, being a library routine, figures to be rather efficient. It is left to the programmer to infer whether something about the program is inefficient and if so, how to optimize it.

UNIX's `gprof(1)`[5], on the other hand attempts to compute a "call graph" profile which, when accurate, attributes the time spent in a procedure *and its descendants* to the procedure. It also attempts to report the fraction of time spent in each child of each function. Unfortunately, `gprof`'s implementation makes the incorrect assumption that every call to a given routine takes the same amount of time. Thus, a `gprof` profile of `process_db` would report half of the time in the first `qsort` call and half in the second. This turns out to be wrong; far more time is spent in the first than the second.

Even when a correct call graph profile is available, as in the Franz Allegro Common Lisp environment[4], it is only a simple special case (length 1 and 2 call paths) of a call path profile (Section 4). One can easily show (by example) that these profiles do not give as much information as a full call path profile. Franz's profiler is based on stack sampling in a way conceptually similar to our own stack sampling (see Section 5.1).

A completely different approach to profiling operates by *tracing* resource usage. `truss(1)` provides a general mechanism for tracing a UNIX command via the `proc(4)` filesystem. The user specifies which classes of resource usage are of interest (page faults, traps, or system calls) and then `truss` prints out a line on the terminal for each page fault, trap, or system call of the selected type that occurs during execution of the trussed process. While it does give a general idea about what resources are being used, tracing generates a large amount of data which typically must be summarized to be understood. For `process_db` above, `truss` simply puts out a line for each of the several thousand page faults encountered, giving little indication of which optimizable context in the program caused the page fault.

The System V version of the SunPro Sparcworks Analyzer[9] provides a rich interface for tracing and profiling program resources. For example, it gives the user a `prof(1)`-like profile of any of the `proc(4)`-accessible resources, gives a graphical display of which text and data pages were touched during execution (including a tool for laying out a program to reduce text page faults), and provides many other features. It even has an option that enables collecting periodic stack samples (the same sampling technique we use in the call path profiler). However, it samples only with respect to the time resource, and instead of a call path profile, abstracts to the less informative `gprof`-style call graph profile.

4 Call Path Profiling

Here is a call path profile of `process_db`:

```
Downward Call Path Profile for resource: Real Time
2676 samples, sampled every 50 gticks
format: usage_fraction (call_path) [#raw_samples]
-----
; 1.00000 (main) [2676]
; 0.88004 (main unifyify_db) [2355]
; 0.68012 (main unifyify_db qsort) [1820]
...
; 0.11323 (main print_salary_list) [303]
; 0.11323 (main print_salary_list extract_salary_fields) [303]
...
; 0.00000 (main print_salary_list qsort) [0]
-----
```

The header information shows that this profile was collected by sampling Real Time once every 50 *generalized ticks* (*gtick*). The *gtick* is a polymorphic unit denoting a hundredth of a second for time resources, one page fault for text and data page faults, one byte for space allocation, etc. Each line after the header shows the resource usage of one *call path* of the program. Intuitively, a call path represents a subset of the program's execution corresponding to a class of program call stacks (Section 4.1 defines this notion precisely). The first field is the fraction of the program's total Real Time attributable to the call path, the second field (in parentheses) names the call path, and the integer in brackets indicates the raw number of samples for the call path.

From this profile, we can conclude that `main` calling `uniquify_db` calling `qsort` consumes 68% of the time, while the other call to `qsort` (within `print_salary_list`) takes insignificant time (no samples hit there). This "where" description directly relates the resource usage to an optimizable context, because the programmer can easily change the program to call a specialized `qsort` within a specialized `uniquify_db` within `main`.

The question remains how to specialize `qsort` in this case. The measurement seems anomalous, since the arrays sorted by the two `qsort` calls have approximately the same number of elements (assuming few duplications in the database). By using `CPPROF` to profile other resources, we find

```
Downward Call Path Profile for resource: Page Faults
9950 samples, sampled every 10 gticks
format: usage_fraction (call_path) [#raw_samples]
-----
0.99889 (main) [9939]
0.71317 (main unifyify_db) [7096]
0.56995 (main unifyify_db qsort) [5671]
...
0.13467 (main print_salary_list) [1340]
0.13457 (main print_salary_list extract_salary_fields) [1339]
...
0.00000 (main print_salary_list qsort) [0]
-----
```

This gives a clear indication of why the first sort takes so much more time: the array to be sorted resides on so many different pages that they can't be held in memory at once, causing a superlinear number of data page faults (due to the superlinear number of key comparisons required by sorting, where each key comparison must actually touch the page to get the name field). Armed with this information, the user can immediately focus on optimizing the critical bottleneck: since the sort key fields are small, the program could localize the keys first (incurring only linearly many page faults) and then sort without causing further page faults. By optimizing `process_db` in this way, we cut the total run time from 25 minutes to about 8. This matches well with what the profile predicts, since we have effectively made the call path 0 cost. The optimization is unnecessary for the call to `qsort` within `print_salary_list`, because the salaries are localized when they are extracted into a separate array.

This illustrates how CPPROF provides a unique combination of highly detailed “where” descriptions with a rich variety of “why” explanations.

4.1 What is a Call Path

A *call path* is a sequence of function pairs $((f_1, g_1), \dots)$ such that f_i executes a subroutine call to g_i and $g_i = f_{i+1}$. In view of the latter condition, we will always abbreviate the call path $((f_i, g_i))$ by $(f_1, f_2, \dots, f_n, g_n)$, as we did in the call path profiles above. By this definition, it is easy to see why a call path corresponds to an optimizable program context: any nested sequence of calls can be rendered separately optimizable by duplicating (with a different name) each subroutine (except for the first) in the list so that it is only called by its predecessor. For example, in our program we could make the call path (main uniquify_db qsort) optimizable by creating qsort_1 and uniquify_db.1 which calls it and altering main to call uniquify_db.1. Note that for recursive programs, one must be careful to maintain an isomorphic call graph by renaming recursive calls to the new name of the recursive function. A general algorithm for *contextualizing a call path* which works for arbitrarily recursive program structures is given in an earlier paper[6]; space constraints prohibit reproducing it here.

4.2 What the Profile Reports

The resource fraction f_p reported for a call path $p = (a \ b \ \dots \ n)$ in a profile has a simple interpretation. Suppose one constructs the optimizable program for p by contextualizing it as above, replacing the call in a to b with a call to b_1 , which in turn calls c_1 , and so on, which finally calls n_1 . We can interpret f_p simply as the fraction of resource usage that would be saved if n_1 were somehow magically transformed to use none of the resource. For example, the time profile for `process_db` predicts that if one were to make `qsort` take no time at all in computing its output when called from `uniquify_db`, then the total run time of the resulting program would be only $1 - 0.68 = 0.32$ as much as the original. This was achieved when we actually carried out the optimization and measured the result.

4.3 Upward and Downward

CPPROF provides two complementary types of call path profiles: *upward* and *downward*. Both are defined in terms of a distinguished subroutine, called the *root* function. In our example above, the root function is `main`, but it can be any function in the program.

Given a root function, the downward call path profile is simply a sorted, filtered list of the resource usages of all call paths whose first function name is the root. (The filtering is based on a user-specifiable *significance threshold*. Any profile entry whose usage fraction falls below the threshold is suppressed from the output.) Both profiles above are downward call path profiles. Downward profiles provide a hierarchical breakdown of resource consumed during the execution of the root function.

An upward call path profile is a sorted, filtered list of the resource usages of all call paths whose *last* function name is the root function. This provides a breakdown of which call ancestors of the root function are incurring the most costs due to it. An important use for this is when one knows that the program is spending most of its time in, say, memory allocation, but doesn't know which calls to it cost the most. `gprof(1)`[5], when its assumption about constant procedure execution times is accurate, provides the information in both upward and downward call path profiles if we restrict call path lengths to at most two.

4.4 A Useful Technique

One useful technique for using CPPROF is to ask first for a *function profile*, which is essentially just a sorted, filtered list of profile entries for all length 1 call paths in the program (i.e., the resource fraction consumed in a function and on its behalf). Next, ask for upward and/or downward call path profiles rooted at functions shown to be significant in the function profile. This assures that one will not be prematurely swamped with a large amount of irrelevant data.

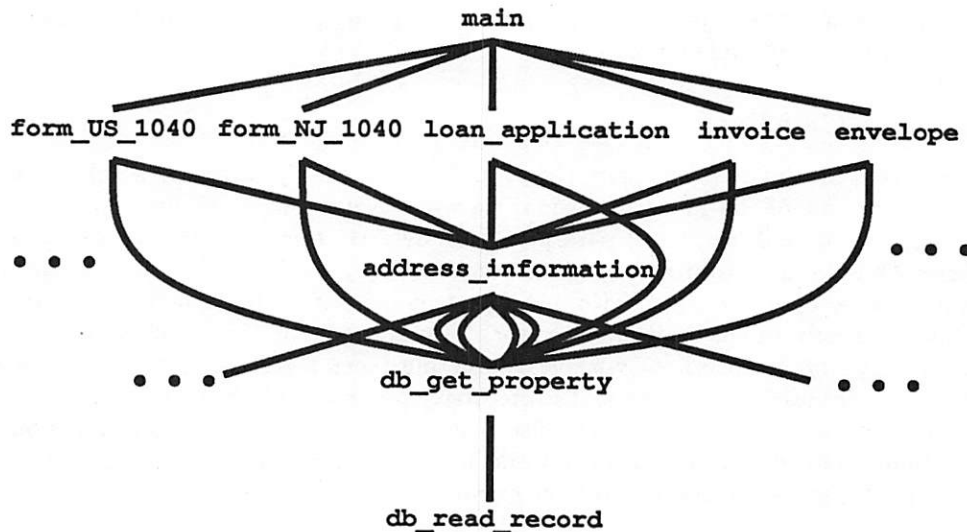


Figure 1: An abstracted call graph for the forms program. *main* calls a specialized processing function for each type of form. These routines call (among others) a utility routine, *address_information*, that returns full name, address, phone, etc., information on an individual in the database. The intermediate utility *db_get_property* extracts a property value from a database entry. The low level routine *db_read_record* performs a disk read to find the record for a given individual. Some subroutines and calls have been left out for brevity.

To illustrate this technique, consider the program shown abstractly in Figure 1. This program fills out forms for users by looking up information in an on-line database. To see why this program is inefficient, we first ask CPPROF for a function profile.

```

Body+Descendants Function Profile for resource: Real Time
  614 samples, sampled every 20 gticks
format: usage.fraction  function.name  [#raw_samples]
-----
1.0000 main [614]
1.0000 db_read_record [614]
0.8632 db_get_property [530]
0.6873 address_information [422]
0.1743 invoice [107]
0.1726 form_US_1040 [106]
0.1726 envelope [106]
0.1726 loan_application [106]
0.1710 form_NJ_1040 [105]
0.1368 db_update_record [84]
-----
  
```

This shows us that the most significant routines are *db_read_record*, *db_get_property*, and *address_information*. At this point, we know that the most significant efficiency gains (if there are any) are to be had in optimizable contexts within these routines. For more specific information, we set the root to the most significant of these, *db_read_record*, and request an upward call path profile for it.

```

Upward Call Path Profile for resource: Real Time
  614 samples, sampled every 20 gticks
format: usage_fraction (call_path) [#raw_samples]
-----
1.0000 (db_read_record) [614]
0.8632 (db_get_property db_read_record) [530]
0.6873 (address_information db_get_property db_read_record) [422]
0.1384 (envelope address_information db_get_property db_read_record) [85]
0.1384 (invoice address_information db_get_property db_read_record) [85]
...
-----

```

This profile directs us to consider first three particular optimizable contexts each ending in a call to `db_read_record`. By examining these in turn, we can discover that the routine `address_information` calls `db_get_property` four times to retrieve four fields of the same database entry. This results in four successive disk reads of the same record. By specializing `db_get_property` for use only within `address_information` (to cache the record after the first read), we can save 75% of the 0.6873 total run time fraction, reducing the overall run time by about 51%. Note that we could not simply optimize `db_get_property` itself, because other calls to it (from outside of `address_information`) require its full generality.

5 Implementation

5.1 Stack Sampling

CPPROF is implemented by a monitor process which controls the target process via `proc(4)` (the time resources are handled slightly differently, using the UNIX interval timers (`setitimer(2)`)). The user specifies which resources are of interest and the monitor process arranges to be notified each time one gtick is consumed. We then implement our own “generalized interval timer” for each resource by counting gticks until the sampling interval² passes. For example, we examine each `read(2)` system call and record the number of bytes requested, each byte being a gtick. Each time a general resource clock passes a sampling interval boundary (e.g. 1024 bytes read), we halt the target process and record a *stack sample* for the resource in a `cprofddata` file. When more gticks pass than a single sample boundary, we credit that many hits to the sample. For example, reading 10240 bytes will result in 10 hits for the corresponding stack sample. A stack sample is simply a list of the number of sample hits, the current program counter, the return address of the current stack frame, the return address of the caller, that of the caller’s caller, etc, until we reach the root procedure (typically `main` for C programs).

5.2 Interactive Profile Analysis

After the target and monitoring processes finish executing, the `cproprof` command analyzes the `cprofddata` file and puts the user in a simple command loop allowing one to view upward and downward call path profiles from user-determined root functions, with user-determined thresholds. Also available is a function profile and a *body profile* (a list of the resource usage consumed in the body of each function; this generalizes `prof(1)` to arbitrary resources). Thus, rather than producing a single static profile, CPPROF is an interactive exploratory environment for understanding the performance of a program empirically.

5.3 Computing Call Path Profiles

We compute a downward call path profile in two steps. First, CPPROF reads the `cprofddata` file once and builds a sample tree. Second, it computes the call path profile for the chosen root from the sample tree. In this section, we explain how to compute a downward call path profile from a given root function for a single resource. Function profiles, upward profiles, and handling multiple resources are all straight-forward extensions of the basic technique.

²To avoid problems with correlation between periodic sampling and periodic program behavior, one can randomly perturb the sampling interval each time.

To build the sample tree, each sample is first converted from a list of addresses into a list of function names by using symbol table information. Once we have the symbolic form of the sample, we insert it into the sample tree as follows. Each node of the sample tree is labeled with a function name, and contains fields for the number of sample hits corresponding to that call stack and a list of children. The root node has the name of the root function, typically `main`. For each sample, we locate the unique path in the tree whose node labels are the same as the function names in the sample (if no such path yet exists, new nodes are installed to create it). The sample hits field of the last node on this path is then incremented by the number of sample hits corresponding to the sample. Once all samples are recorded in the tree, we compute for each node in the tree the sum of its sample hits and those of all its descendants; this number is termed the "weight" of the node.

In the special case of a nonrecursive program for which the user's choice of root function is the same as the label of the root node of the sample tree, the resource fraction for any call path may be read directly from the sample tree. One locates the unique path in the sample tree labeled with the same list of function names and divides the last node's weight by the weight of the root node.

To handle the general case, including recursive programs and non-main root functions, we use the more general algorithm described in the Appendix.

5.4 Portability Issues

The great majority of code for CPPROF is completely portable, ANSI C. One small module, the stack trace sampler, is architecture- (and sometimes compiler-) dependent. Stack and register conventions will vary across machines; however, once these conventions are understood, walking the call stack using the `proc(4)` interface is straight-forward to implement.

Obviously, the part of CPPROF that monitors via `proc(4)` is portable only among UNIX System V systems. A limited version of CPPROF (time only) is implemented under BSD UNIX. It has also been implemented via stack sampling for time and space resources in Franz Allegro Common Lisp v4.1 and also under Macintosh Common Lisp v2.0.

Thus, it appears that the technique generalizes easily to other operating systems and machines, limited only by how well the environment supports access to resource data. System V's `proc(4)` appears to be the richest environment so far. It could, we believe, be further improved by adding a "generalized interval timer" feature (analogous to the time version), whereby instead of receiving a signal on *every* resource usage, the user could set an interval (in terms of gticks) at which to be signaled. This would allow monitoring with even less overhead, avoiding the context switch necessary to handle each resource signal.

6 Experience/Case Studies

While the System V implementation of CPPROF is still in testing, we have nevertheless applied it to several large programs, including six of the Spec benchmarks. Though we are not familiar enough with the benchmark programs to do detailed optimization, an example of what can be found quickly by call path profiling is illustrated below.

```
Downward Call Path Profile for resource: User Time
1871 samples, sampled every 20 gticks
format: usage_fraction (call_path) [#raw_samples]
```

```
-----
1.00000 (xleval) [1871]
1.00000 (xleval evform) [1871]
1.00000 (xleval evform xprog) [1871]
1.00000 (xleval evform evfun) [1871]
...
0.27579 (xleval xlgetvalue) [516]
...
0.09139 (xleval evform xlevlist consa) [171]
...
-----
```

This is a downward call path profile for user time of the 022.1i Spec benchmark, a Lisp interpreter (itself coded in C) solving the 9-Queens problem. Since the code implements an interpreter, one expects to find inefficiency compared to running compiled code. Given an understanding of the structure of the program, the call path profile highlights these inefficiencies. For example, the call path (`xlevel evform xlevlist consa`) uses 9.1% of the total run time. Looking at the code, the `xlevlist` function is the interpreter's routine for evaluating the arguments to a Lisp function call and returning a list of the evaluated actuals. `consa` allocates a Lisp cons cell. Thus, this profile entry reports that simply consing up the list of function call actuals (not including the time to evaluate each actual) is using 9.1% of the total time. Compiled code would avoid this inefficiency by passing values at fixed locations on the stack. Note that a `prof(1)` profile does not show this at all, because most of the work done by `consa` is done in its call descendants. Also, there is more than one caller of `consa` in the program, so we could not simply have deduced this information from a function profile alone.

In addition to testing on the Spec benchmarks, Lisp-based implementations of the call path profiler have been used on several large programs to find significant optimizations. In one case (involving a system of several thousand lines of Lisp source code), it exposed the fact that one of the program modules had mistakenly been left uncompiled, hence was run interpreted. This was shown simply by the presence of "extra" calls in call paths to stack blocks established by the interpreter that are normally compiled out. Simply compiling the module resulted in a 10% savings in total run-time.

In another large application, call path profiling detected the fact that redundant calls to a data structure canonicalization procedure slowed the program significantly, pointing out the need to canonicalize only when necessary.

7 Discussion

Limitations. The sampling approach to profiling implemented in virtually all profilers to date has several limitations, centering around statistical significance and unwanted correlations between the sampling and the executing code. While time sampling using `setitimer(2)` suffers from these problems, sampling other resources fares better. By randomizing the general timer interval, we can avoid the correlation problem, and by reducing the mean of the timer interval, we can increase the statistical significance as much as desired (though this may further perturb the program's unsampled behavior).

Call path profiling implemented by sampling has some other limitations. As discussed in a previous paper[6], a sampled call path profile can only give information on *dynamic* call paths, because we sample the actual call stack at run time. The user is actually interested in *lexical* call paths, i.e. those in the source code, because those are the ones that are directly optimizable. In most cases, the two sets of call paths are the same, but they can differ in programs where a function is stored or passed as a first class data object and actually called in a context different from where it was created or referred to. For example, a call path profile may report a call path having a call from *f* to *g* when in the program text there is no explicit call of *g* from *f*; instead a pointer to *g* is passed somehow into *f* and is then called. Ideally, one would like the call path to reflect the lexical originator of *g*, instead of the run-time caller. This problem is fundamental to the sampling-based approach and can only be solved by instrumenting the source code.

Another limitation of our implementation is that we have ignored the fact that one function may call another from more than one call site in its body. Ideally, the user would like to see call paths that distinguish the different call sites, as different call sites can have widely different costs. It is straightforward to incorporate this into `CPPROF`, because the stack samples are recorded as program addresses. We simply have not yet added this level of detail to the prototype.

Other Relevant Work. Some profilers, such as the Franz Common Lisp graphical profiling tool[4] (time and space resources) and the Plan 9 profiler[1] (time only), collect and display a tree-based, hierarchical representation of what we called the sample tree: time spent in individual program stack traces. We term this a *sample tree profile*. For non-recursive programs, this gives the same information as a downward call path profile rooted at the main routine; however, the call path profiler provides better

summarization and attention focusing tools for complex programs. For example, a sample tree profile of the *forms* program of Figure 1 looks like this:

```

Sample Tree Profile for resource: Real Time
614 samples, sampled every 20 gticks
format:frame_designator (usage_fraction) [#raw_samples]
-----
main (1.0000) [614]
  invoice (0.1743) [107]
    address_information (0.1384) [85]
      db_get_property (0.1384) [85]
      db_read_record (0.1384) [85]
    db_get_property (0.0358) [22]
    db_read_record (0.0358) [22]
  loan_application (0.1726) [106]
    address_information (0.1368) [84]
      db_get_property (0.1368) [84]
      db_read_record (0.1368) [84]
    db_get_property (0.0358) [22]
    db_read_record (0.0358) [22]
  envelope (0.1726) [106]
    address_information (0.1384) [85]
      db_get_property (0.1384) [85]
      db_read_record (0.1384) [85]
    db_get_property (0.0342) [21]
    db_read_record (0.0342) [21]
  form_US_1040 (0.1726) [106]
    address_information (0.1368) [84]
      db_get_property (0.1368) [84]
      db_read_record (0.1368) [84]
    db_get_property (0.0358) [22]
    db_read_record (0.0358) [22]
  form_NJ_1040 (0.1710) [105]
    address_information (0.1368) [84]
      db_get_property (0.1368) [84]
      db_read_record (0.1368) [84]
    db_get_property (0.0342) [21]
    db_read_record (0.0342) [21]
  ...
-----

```

We have used indentation to represent depth in the tree. Even though this mass of data is complete, in the sense that any other profile can be derived from it, it does not focus one's attention on worthwhile optimization opportunities. Nothing looks very significant, because every node below *main* has a usage fraction less than 0.2. In particular, every entry for *db_read_record* is less than 0.14. In order to find that *db_read_record* itself accounts for essentially all the time in the program, one must add up fractions that are widely distributed over the tree. Furthermore, to find the usage fraction for any optimizable context not rooted at *main*, one must perform a similar addition. The call path profiling methodology discussed in Section 4.4, on the other hand, leads the user directly to considering the three most worthwhile optimizable contexts.

Future Work. We believe call path profiling of general resources should be incorporated into any profiling environment that is sophisticated enough to record stack samples, such as Sparcworks[9] or the Franz profiler[4]. This involves two extensions. First, the environment must add the capability to sample monotonic resources on an interval basis, hopefully with kernel support. Second, the environment must provide an interactive capability to compute and display call path profiles based on user-determined root functions and thresholds. While the first extension may be somewhat involved, the second could be implemented quite quickly for whichever resources are already supported, such as elapsed time or allocated space.

Conclusion. Call path profiling provides an informative description of where a program consumes its resources. It is closely tied to optimization, because it reports the costs of optimizable program contexts, rather than simply the context-free costs of function bodies. This technique for describing “where” is equally applicable to any monotonic resource (such as time, space, page faults, traps, and system calls), so by profiling via System V’s `proc(4)` filesystem, CPPROF is able to answer the “why” profiling question in a richly detailed way as well.

Our research indicates also a way in which `proc(4)` (or any process monitoring facility) could be improved: provide kernel support for “generalized interval timers.” That is, allow the user to select a sampling interval for each type of resource and only signal the monitoring process as each interval elapses. The current `proc(4)` signals on every resource `gtick`. This can needlessly increase the monitoring overhead of a run, both increasing the overall execution time of the profile run and increasing the uncorrectable perturbation to the program’s true behavior (due to extra context switches, cache flushes, etc). On the other hand, adding this general service to the kernel should be inexpensive and simple to do (we have implemented the general interval timers at the user level, but one expects that support at the kernel level would sharply reduce the overhead of the mechanism).

We believe that too few programmers profile their programs. Perhaps this is due, in part, to the difficulty in connecting previous forms of profiler output data with optimizable program contexts. We hope that call path profiling can remove much of the detective work previously required, making profiling more attractive to programmers.

Appendix

This appendix presents pseudo-code for a general algorithm for computing downward call path profiles from a previously constructed sample tree. It works for arbitrarily recursive programs and for arbitrary choice of root function by the user. A separate output routine takes the output of this program and formats it for readability; in particular, it divides the number of sample hits in each call path’s profile entry by the total number of sample hits incurred during the run in order to report a resource usage fraction.

Call paths are represented as lists of strings, each string being a function name. The functions *cons*, *first*, and *rest* operate on lists: *cons* constructs a new list by adding an element at the beginning of an old list; *first* returns the first element of a list; and *rest* returns the list obtained by removing the first element of a list.

The basic idea of the algorithm is to recursively descend the sample tree and associate each node with a canonical call path that represents it. Then, we add the weight of the node to the ticks field of that call path’s record, unless the record is locked. We have to use a locking protocol in the presence of recursion to avoid attributing the same ticks to a given call path multiple times.

Let PATH-HASH be a hash table whose keys are call paths
and whose values are records containing a *ticks* field
and a *locked?* bit

Procedure **Downward-CP-Profile** (SAMPLE-TREE, ROOT-FN)
(Returns a sorted list of call path resource usage records)
Initialize PATH-HASH to be empty.
Summarize-below-root(SAMPLE-TREE, ROOT-FN)
Let RESULT be a list of every (key, value) pair
in PATH-HASH.
Sort RESULT in decreasing order of sample hits field.
Return RESULT.

Procedure **Summarize-below-root** (TREE-NODE, ROOT-FN)
If *name*(TREE-NODE) = ROOT-FN
Then **CPP-Rec**(TREE-NODE, ())
Else For each child CHILD of TREE-NODE
 Summarize-below-root(CHILD, ROOT-FN)

Procedure **CPP-Rec**(TREE-NODE, PARENT-CALL-PATH)
Let NAME = *name*(TREE-NODE)
CP = *cons*(NAME, PARENT-CALL-PATH)
NEXT-PARENT = **Canonicalize-name**(CP, NAME)
RECORD = *get-record*(CP, PATH-HASH)
LOCKED? = *record-locked?*(RECORD)
If not LOCKED?
 Then *lock-record*(RECORD)
 add-ticks(*weight*(TREE-NODE), RECORD)
For each child CHILD of TREE-NODE
 CPP-Rec(CHILD, NEXT-PARENT)
If not LOCKED?
 Then *unlock-record*(RECORD)

Procedure **Canonicalize-name**(CALL-PATH, THIS-NAME)
For (PTR = *rest*(CALL-PATH); While PTR is not null)
 If *first*(PTR) = THIS-NAME
 Then Return PTR
 Else Set PTR = *rest*(PTR)
Return CALL-PATH

Acknowledgements

The authors would like to thank Mark Plotnick for his assistance. Thanks also to Rob Pike and Ken Thompson for stimulating discussions of this work.

References

- [1] *The Plan 9 Programmer's Manual*. Murray Hill, NJ: AT&T Bell Laboratories. (1993).
- [2] R. Faulkner & R. Gomes, The process file system and process model in UNIX system V, In *Proc. Winter 1991 USENIX Conference*. Berkeley, CA: USENIX Assoc. (1991) 243-252.

- [3] P. Flajolet, B. Salvy, & P. Zimmerman, Automatic average-case analysis of algorithms, *Theoretical Computer Science* 79, pp 37–109. Amsterdam: Elsevier Science Publishers, 1991.
- [4] Franz, Inc., *Allegro COMPOSER User Guide, version 1.0*, Chapter 6. Berkeley, CA: Franz, Inc, 1990.
- [5] S.L. Graham, P.B. Kessler, & M.K. McKusick, Gprof: a call graph execution profiler, *ACM SIG-PLAN Notices* 17(6), pp 120–126. June, 1982.
- [6] R.J. Hall, Call path profiling, In *Proc. 14th International Conf. on Software Engineering*, New York, NY: Assoc. for Computing Machinery, (1992), 296–306.
- [7] D.E. Knuth, An empirical study of FORTRAN programs, *Software-Practice and Experience* 1, pp 105–133. 1971.
- [8] D. Le Metayer, ACE: An Automatic Complexity Evaluator, *ACM Trans. on Programming Languages and Systems* 10(2), pp 248–266. 1988.
- [9] SunPro, *Performance Tuning an Application*. Sun Microsystems, Inc. (1992).

Robert J. Hall (hall@research.att.com) has been a Member of Technical Staff at AT&T Bell Laboratories, Murray Hill, since 1991. Prior to that, he received his S.M., E.E., and Ph.D. degrees from the Massachusetts Institute of Technology, where his thesis research was performed under Charles Rich in the Artificial Intelligence Laboratory's Programmer's Apprentice Group. He received the B.S. degree *summa cum laude* in E.E.C.S. from the University of California at Berkeley.

Aaron J. Goldberg (aaron@research.att.com) has been a Member of Technical Staff at AT&T Bell Laboratories, Murray Hill, since 1992. Prior to that, he received his Ph.D. degree from Stanford University, where his thesis research was performed under John Hennessy in the DASH group. He received the A.B. degree *summa cum laude* in Applied Mathematics from Harvard University.

Computer System Performance Problem Detection Using Time Series Models

Peter Hoogenboom and Jay Lepreau

University of Utah

Abstract

Computer systems require monitoring to detect performance anomalies such as runaway processes, but problem detection and diagnosis is a complex task requiring skilled attention. Although human attention was never ideal for this task, as networks of computers grow larger and their interactions more complex, it falls far short. Existing computer-aided management systems require the administrator manually to specify fixed "trouble" thresholds. In this paper we report on an expert system that automatically sets thresholds, and detects and diagnoses performance problems on a network of Unix computers. Key to the success and scalability of this system are the time series models we developed to model the variations in workload on each host. Analysis of the load average records of 50 machines yielded models which show, for workstations with simulated problem injection, false positive and negative rates of less than 1%. The server machines most difficult to model still gave average false positive/negative rates of only 6%/32%. Observed values exceeding the expected range for a particular host cause the expert system to focus on that machine. There it applies tools with finer resolution and more discrimination, including per-command profiles gleaned from process accounting records. It makes one of 18 specific diagnoses and notifies the administrator, and optionally the user.¹

1 Introduction

Existing computer and network management tools require the administrator manually to specify fixed threshold values of performance or load criteria. This paper describes a time series modeling technique that can be used for more automatically detecting computer system performance problems. The technique is based on an *exponentially weighted moving average* time series model. It allows detection of performance problems in a host by providing a means of detecting performance criteria values that are out of normal ranges. The effectiveness of the technique is demonstrated by its use in the **System Performance Advisor (SPA)** system administration expert system. The purpose of SPA is to assist a Unix² system administrator in *system performance management*. We define system performance management to be those activities performed by a system administrator to ensure a computer system is providing as much of its capacity as possible to its users. SPA does this by monitoring hosts and processes in a network. Problems with hosts and processes make themselves apparent as a deviation from normal activity levels. After a problem is detected, SPA alerts the system administrator.

Performance can be defined and measured in many ways. Typically, the system administrator chooses system metrics that are easily obtained through standard utilities. Measurements such as CPU utilization, memory utilization, paging rate, and load average are commonly used.

The process of managing a system's performance is an iterative one. Performed by a human, the steps involved might not be as well-defined as described here, but the basic iterative steps still exist. The performance management process is composed of a *definition* phase, a *diagnosis* phase and a *therapy*

¹This research was supported in part by the Hewlett-Packard Research Grants Program and the University of Utah.

²Unix is a trademark of AT&T Laboratories.

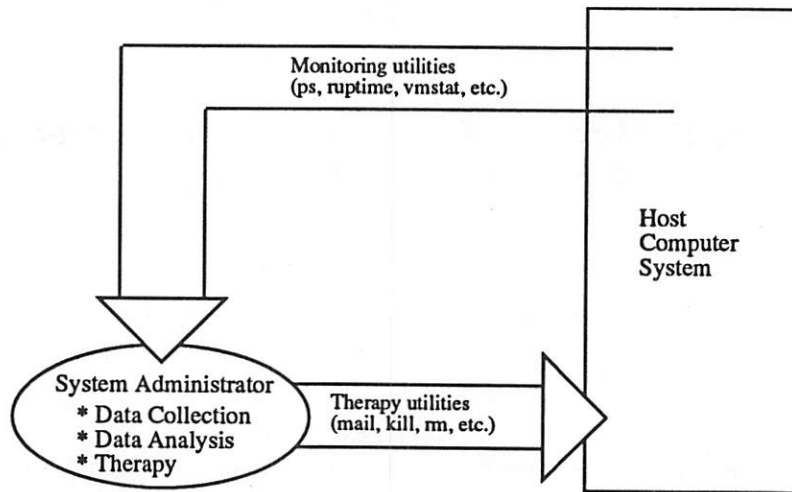


Figure 1: The Traditional Process of System Performance Management

phase. The definition phase consists of determining what performance criteria are to be evaluated and how the data will be collected and analyzed. Diagnosis begins by making measurements of the system. This is followed by analysis of the collected data. Analysis is done manually or with assistance from another computer. During analysis, the performance problems are recognized and categorized. After the analysis, if the results suggest that some improvements in performance are possible, a therapy phase is begun. During this phase, adjustments to the system are made in an attempt to alleviate the problem. The remaining phases are repeated until there are no known performance problems in the system.

2 Why is System Performance Management Difficult?

Traditionally, system performance management activities are carried out by human system administrators who rely on their own expertise to ensure maximum performance of all systems. As shown in Figure 1, the system administrator monitors the host computer system, analyzes the collected data, and performs a therapy procedure on the host as needed. In addition to this process being very time-consuming, there are several difficulties with this approach [13]:

- frequently, the best guesses by system administrators are wrong; and
- performance degradations of up to 20% go unnoticed on a regular basis.

In addition, regardless of the problem domain, human expertise suffers from several other limitations:

- it is not always available (illnesses, vacations, etc.);
- it suffers from inconsistencies due to skill level, emotional state, and attentiveness; and
- it is not scalable. For example, suppose a human system administrator can consistently and objectively manage the system performance of all users and programs on 25 processors. Can the same system administrator effectively manage all users and programs on 250 processors?

An expert system approach to managing system performance (shown in Figure 2) can reduce or eliminate these disadvantages. It is always available, it applies its expertise consistently, and it is scalable. Since the human's expertise relies on remembering what a given host, user, or program has done in the past, the more hosts, users, and programs he has to deal with, the less knowledge he has on an individual host, user, or program. This does not happen with an expert system approach.

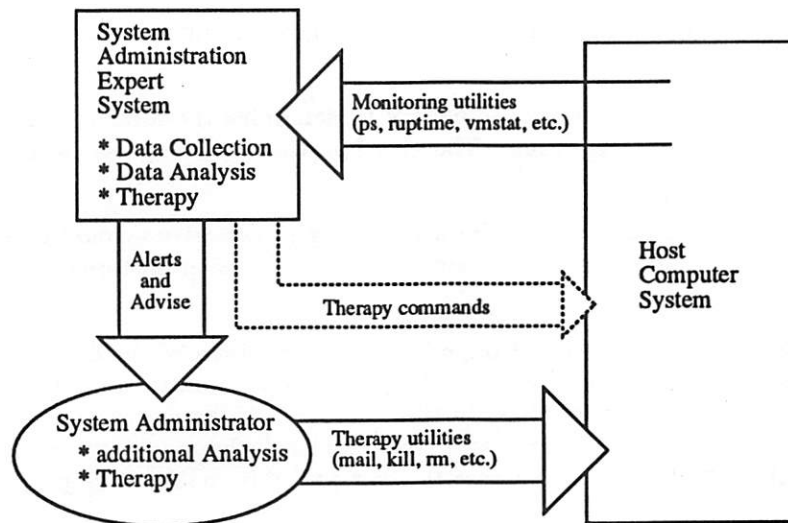


Figure 2: The Expert System-based Process of System Performance Management

3 Developing a Time Series Model of Computer System Performance

Before developing the SPA expert system, we found it necessary to define a mechanism that could reliably indicate normal system performance and the expected amounts of deviation from that performance level. Setting fixed threshold levels for problem detection, as today's commercial network management systems require, was deemed unacceptable. Here are several realities of computer system usage that make it difficult to define a single, static model of "expected" behavior that is appropriate for all hosts in the network, or even setting a different one for each host (tedious as that might be):

- Each processor (host) can have a speed and configuration different from other processors on the network.
- The expected workload distribution is different on each processor.
- The expected workload distribution on each processor varies over time. Variations can occur daily, weekly, or over longer periods of time.
- Each user on the system wants to accomplish different types of tasks. Users that use the same program do so in a way that could be unique to the user.

For these reasons, SPA must maintain a different model of the expected utilizations and activity levels for each host. Several possible system utilization modeling techniques were investigated.

3.1 Modeling Alternatives

The purpose of maintaining a model of hosts and processes is to be able to, at time t , make a forecast of what the behavior will be at time $t + 1$. If the forecast error (the difference between the forecast and the actual value at time $t + 1$) is significant, there is a performance problem to be investigated.

There are several ways to model hosts and processes. The modeling techniques discussed below are taken from different branches of mathematics, but all have been extensively studied. This provides a rich mathematical background on which to base the modeling technique chosen for SPA.

The successful modeling technique must satisfy the following criteria:

Forecast Quality: The model must be able to accurately predict future behavior. No modeling technique is able to exactly predict future behavior, but some modeling techniques are better at modeling certain

kinds of behavior. A modeling technique should be chosen to minimize forecast errors as much as possible.

Time Efficiency: The model provides a simple way to determine the forecast, the forecast error, and whether the forecast error is significant. The model provides a "best guess" at time t of the behavior at a future time $t + 1$.

Space Efficiency: The model does not require maintaining an extensive amount of history information. Less history data means less space used to store the model, but some history data is probably required to maintain forecast quality.

Responsiveness: It is able to deal with changes in the characteristics of the data being modeled. Much of the data being considered undergo some kind of seasonal variation. The model should be able to modify itself automatically to adjust to these changes. If the user were required to make these adjustments on all the hosts and processes being modeled, the models would quickly become out-of-phase with the actual behavior, because the user probably could not keep up with the necessary changes.

Configurability: If necessary, the system administrator should be able to modify the behavior of the model to improve its performance. This is also known as *tuning* the model.

Interpolation Techniques

The data collected by SPA are implicitly a function of time. For example, suppose there exists a series of n values y_1, y_2, \dots, y_n collected at n distinct times t_1, t_2, \dots, t_n . The collected values can be described by a function $Y(t)$. Thus,

$$y_1 = Y(t_1), y_2 = Y(t_2), \dots, y_n = Y(t_n).$$

The model should be able to determine what value should be expected at time t_{n+1} . That is, what is $Y(t_{n+1})$? If a function $Y(t)$ can be determined that completely describes the data collected so far, it can be used to find out $Y(t_{n+1})$. It can be shown [14] that given n data points, a unique polynomial of degree $n - 1$ can be found which has the desired property of satisfying the equation shown above.

Polynomial Interpolation and *spline interpolation* are common approaches to this. Both techniques suffer from similar problems that reduce their likelihood of being used for system performance analysis. First, interpolation is very effective in fitting a curve to known data points, but not so effective for forecasting a future reading, even if the time at which the reading takes place is very close. Predictive abilities are worse when the data points are partly stochastic in nature. Second, the mathematics for even one model is complex. Polynomial interpolation is $N \cdot (\log N)^2$ for advanced methods and quadratic for simpler techniques [14]. Although a cubic spline on n points can be constructed in linear time, few, if any, computer system performance criteria are modeled accurately with a cubic equation. Third, extensive history information is required. One could reduce the amount of required history by only using the last N readings, but this is at the cost of forecast quality.

Curve Fitting

This approach is closely related to that of interpolation discussed in the previous section. The difference with this approach is the admittance of a stochastic element in the data. Thus, $Y(t)$ does not exactly describe all the points y_1, y_2, \dots, y_n . However, the "best" (least amount of error) $Y(t)$ is still desired. Numerous methods of minimum discrepancy exist to fit data to a linear function (commonly known as the *method of least squares*) and more complex functions. Periodic functions such as those necessary to model daily, weekly, and quarterly variations in workload on a computer system can be approximated accurately using fast Fourier transform techniques.

Unfortunately, the mathematics is complicated and time-consuming for techniques more advanced than least squares fit for a linear function. The network in which SPA operates has 100–200 hosts that are

used by hundreds of users. There are dozens of supported programs on each of these hosts. To implement models using these techniques does not allow this level of scalability. Another disadvantage is the slowness of response to changes: no allowances for a seasonal variation are made.

Queueing Theory

Queueing theory is a branch of mathematics that deals with situations involving waiting for access to a shared resource. There are many applications for a theory of queues including factory assembly lines, highway traffic flow, jet traffic near an airport, a checkout counter at a supermarket, and, not surprisingly, computer systems.

Discussions of queueing theory usually refer to the objects waiting in line as *customers* and to what they are waiting for as a *service center*. The description and analysis of arrivals of customers and the delay they encounter waiting for service is the object of study in queueing theory. Typical quantities that are provided from a queueing model are the average number of customers in the queue and the average time a customer spends in the service center.

One attractive feature of a queueing model is that it provides a modeling technique in which the model is expressed in terms similar to the problem domain. Another attractive feature is that the use of queueing theory for computer system performance evaluation has already been extensively studied and reported. Most notably are the books by Ferrari, Serazzi, and Zeigner [4] and Lazowska, Zahorjan, Graham, and Sevcik [7].

There are several problems with using queueing models for the purpose of real-time problem detection. First, it is not clear from the queueing theory literature how one determines the parameters of the queueing model from historical data. Second, after they are initialized, a queueing model is not responsive to changes in the characterizations of workload. Third, after the model is in use, updating its parameters to reflect a changed workload must be done manually.

Probability Distributions

Probability distributions are employed when the variable of interest is continuous or nearly continuous. For a continuous variable, the probability of encountering any one value is of little interest because the probability is zero when the number of possible values is infinite. Thus, the probability of a variable being in a certain range of values is more interesting. Figure 3 shows a typical frequency distribution of a time series for load average measurements. Detecting problems using this type of model is a matter of deciding that the probability of a given value occurring is low enough to be considered unusual.

In the case of Figure 3, the probability of a load average measurement being higher than, for example, 10.0 can be calculated. The historical data for the host provide a total number of samples taken and how many of those are values above 10.0. From this information, the probability of an individual load average being above 10.0 can be calculated.

There are several disadvantages to this approach. First, a large number of samples must be collected and reviewed in order to determine a probability distribution that is reasonably accurate. Second, in order to account for workload variations, the distribution must vary over time. This could be approximated by having a different distribution for each hour of the day. To account for daily and weekly variations on N hosts requires $N \cdot 7 \cdot 24$ distributions. Third, the range of values possible and the number of intervals must be known beforehand. If not, the system must dynamically recalculate the distribution ranges, interval count, and probabilities. This could be computationally expensive if the recalculation occurs frequently.

Time Series Models

Time series analysis is based on the assumption that the data collected varies as a function of time. This is certainly true for load average measurements. The idea behind time series analysis is to examine the

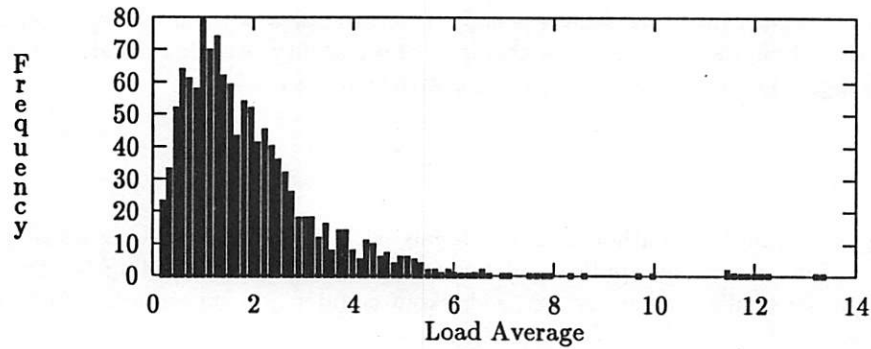


Figure 3: Typical Load Average Frequency Distribution

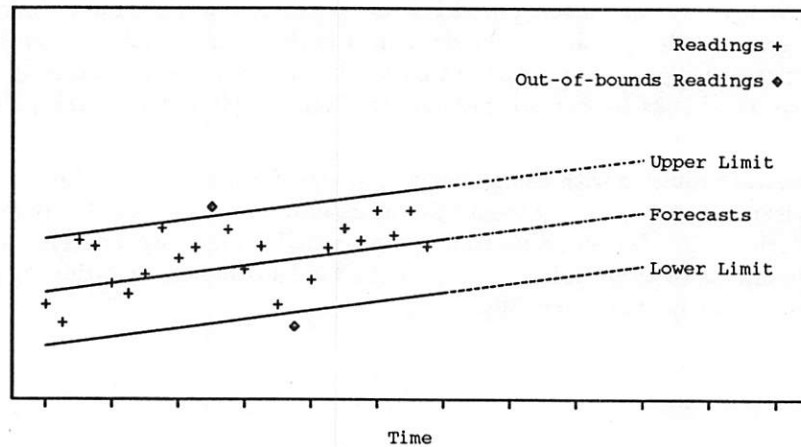


Figure 4: A Linear Time Series Forecasting Model

history of data collected and do one or more of three things: recognize patterns in the history data, calculate statistical measures of the history data, or forecast what the time series will do at some future time.

Figure 4 shows a forecasting model that recognizes readings as abnormal when they are more than a fixed amount from the forecast. In this example, the forecasts are described by the equation of a line, but could also be the current mean of the readings. The upper and lower limits could be one standard deviation from the mean.

By maintaining a history of readings collected from the system, useful statistical measures such as mean, standard deviation, minimum, and maximum can be calculated. This type of time series model is called a *constant model* because it can be expressed algorithmically as

$$x_t = m + \epsilon_t$$

where x_t is the forecasted value at time t , m is the mean, and ϵ_t is a random component that is assumed to have mean 0 and variance v_ϵ . One difficulty with this model is dealing with variation in the modeled object over time. The more readings the history contains, the slower the mean responds to changes in the process and the more costly it is to calculate.

One way to speed the response and calculation of the mean is by using a *moving average of length k* [11]. The moving average is calculated by considering the last k readings only. The moving average model does handle a seasonal variation in the data; however, it requires a fixed number of readings in each season to accomplish this. Also, if the season is long, a large number of readings must still be kept to calculate the moving average.

For data values that are known to experience seasonal variations, a *seasonal model* can be employed. The seasonal model is also known as an *exponentially weighted moving average* time series model [17]. Seasonal models arise frequently in time series analyses of business and economic data. In those types of studies, the readings often fluctuate according to the seasons and business cycles. Often it is desirable to remove the effect of these fluctuations to study the readings without the influence of a particular season. Other times, a forecast of what to expect in the next time period is sought. An accurate forecast requires a model that can account for the seasonal variation. The seasonal model also overcomes the deficiencies of the moving average model: any number of readings can be taken and once the model is initialized, no history data values are required.

In a seasonal time series model, the values being modeled have four components: *constant*, *trend*, *seasonal*, and *random*. The model can be used to account for the first three of these components. The constant component is the portion of the data that is always present. The trend component reflects the fluctuation in the data that extends throughout the entire time series. For example, the utilization of a computer system might increase from one year to the next as more people use it. The seasonal component reflects the regular variations that occur over a specific period of time. For example, the daily variation in workload readings. Usually, they are higher during the day than at night. Finally, the random component accounts for fluctuations in the data due to undetectable causes.

The basic form of the seasonal time series model is written as

$$x_t = b_1 + b_2 t + c_t + \epsilon_t$$

where b_1 is a constant component, b_2 is a trend component, c_t is a seasonal factor, and ϵ_t is a random error component. The effect of the seasonal factors c_t is to de-seasonalize the current reading x_t . The length of the seasonal variation is fixed at length L . In the case of a load average time series for a host, $L = 24$ hours.

Because b_1 , b_2 , and $c_t, t = 1, \dots, L$ cannot be determined exactly, they must be estimated. These estimates are updated at the end of each of the L periods. Thus, these estimates respond quickly to any changes in the profile of the data.

The model adapts to changes in the data by the use of three *smoothing constants*: α , β , and γ . The usage of these smoothing constants is analogous to the usage of the decay rate in the calculation of load averages in Unix operating systems [8]. The α , β , and γ smoothing constants are used to smooth the constant, trend, and seasonal components of the time series model, respectively.

A forecast of a data value at some time $t + k$ in the future is computed from:

$$x_{t+k} = \hat{b}_1(t) + \hat{b}_2(t) \cdot k + \hat{c}_{t+k}(t+k-L)$$

where $\hat{b}_1(t)$ and $\hat{b}_2(t)$ are the estimates of $b_1(t)$ and $b_2(t)$ at the end of the time period t , and $\hat{c}_{t+k}(t+k-L)$ is the estimate of the seasonal factor for period $t+k$ that was computed L periods ago.

The estimates \hat{b}_1 , \hat{b}_2 , and $\hat{c}_t, t = 1, \dots, L$ are computed as follows:

$$\begin{aligned} \hat{b}_1(t) &= \alpha [x_t - \hat{c}_t(t-L)] + (1-\alpha) [\hat{b}_1(t-1) + \hat{b}_2(t-1)] \\ \hat{b}_2(t) &= \beta [\hat{b}_1(t) - \hat{b}_1(t-1)] + (1-\beta) [\hat{b}_2(t-1)] \\ \hat{c}_t(t) &= \gamma [x_t - \hat{b}_1(t)] + (1-\gamma) [\hat{c}_t(t-L)] \end{aligned}$$

where $0 < \alpha, \beta, \gamma < 1$.

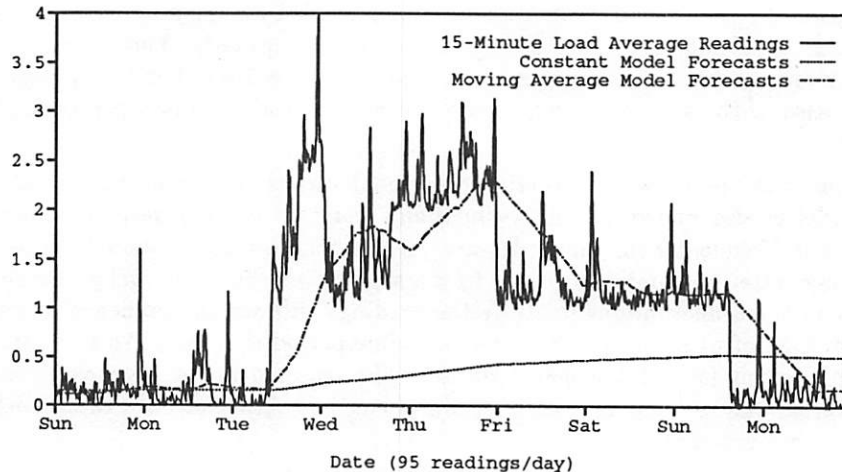


Figure 5: Constant Model and Moving Average Model of 15-Minute Load Average

The calculation of the initial estimates $\hat{b}_1(0)$, $\hat{b}_2(0)$, and $\hat{c}_t, t = 1, \dots, L$ can be done from historical data using a least-squares linear regression [11], or by simpler methods[9]. After the initial estimates are computed, the model is used with no further need to reference the historical data. The smoothing constants α , β , and γ are determined heuristically by the model developer. Smaller values for the smoothing constants give more weight to previous readings. This results in the model responding more slowly to changes in the time series. Larger values give more weight to recent values. Thus, the model reacts more quickly to changes. Very large values are to be avoided, however, since the model will over-react to random fluctuations.

The seasonal model described above is one of a family of seasonal models based on a method originally described by Winters [17]. The model and notation described above is from Montgomery, Johnson, and Gardiner [11].

Comparison of the Three Time Series Models

Three varieties of time series models have been described. How well the model performs depends on the values being modeled. An example of the usage of these three models on a series of 15-minute load average readings from jensen.cs.utah.edu is given in Figures 5 and 6. The constant model and moving average model (length is 95 readings) are shown in Figure 5. The seasonal time series model is shown in Figure 6. This host is a HP 9000/380 with 32 MB of real memory. It serves mainly as a fileserver, but also receives moderate usage interactively during the day and in compiling large software systems at night. The displayed values are one portion of 3 weeks of load averages collected. This portion of the data shows the host experiencing long periods of time in which the load averages are always above 1.0. The constant model is clearly unresponsive to the changes in the workload characterization. This is due to the long past history of readings. The moving average model provides much better response: forecasts errors are within tolerable limits within a day. The time series model provides even better response. The standard deviation of forecast errors was 0.6716 for the constant model, 0.4316 for the moving average model and 0.1077 for the seasonal model.

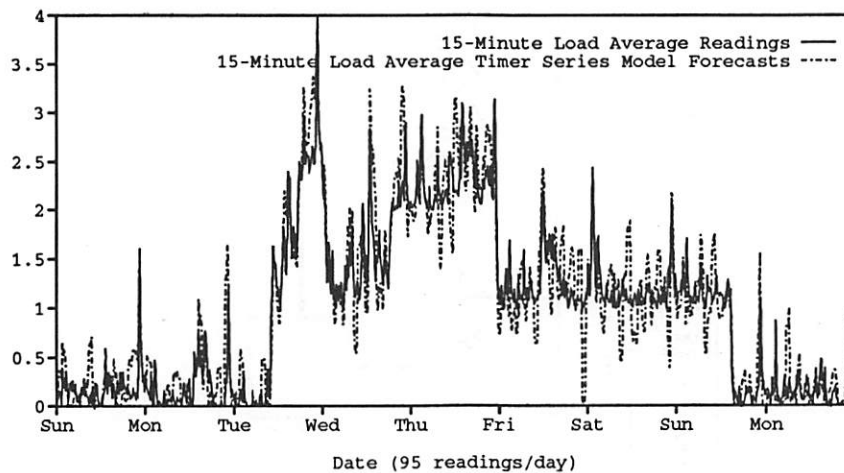


Figure 6: Seasonal Time Series Model of 15-Minute Load Average

4 The SPA Expert System

The SPA expert system was developed as a means of validating the use of time series models in detecting computer system performance problems. It consists of 10,800 lines of Common Lisp code and two small C language programs. The development of the Common Lisp portion of SPA, hereafter referred to simply as SPA, was accomplished using the FROBS system [12] as an expert system shell. This system provides a merger of two common knowledge representation techniques: object-oriented programming and frame-based data structures. The programmer can choose the technique that seems most suited to the problem at hand. In the case of SPA, FROBS forward-chaining rules and frame data structures³ can be chosen to express data-driven reasoning about hosts and processes or object-oriented programming techniques can be chosen to express data abstraction and build hierarchies of objects in the knowledge base.

The forward-chaining rules provide SPA the ability to reason and make decisions about the knowledge base. A forward-chaining rule can be thought of as an *IF condition THEN action* statement. If *condition* (also called the *premise*) is true in the knowledge base, then the rule is said to *fire*, and the specified *action* (also called the *conclusion* of the rule) is performed. The specified *action* can have side-effects in the knowledge base which might cause the firing of additional forward-chaining rules.

SPA's forward-chaining rules describe a relationship between what is currently true about the knowledge base and what actions SPA should take. This, along with the ability to dynamically add and remove rules from the system give SPA a considerable advantage over system administration tools written in a compiled, procedural language. Experience with expert systems has shown it normally impractical to encode and tune the large required knowledge bases, when expressed strictly procedurally.

A functional description of the SPA expert system and its major software components is shown in Figure 7. The Common Lisp code contains the knowledge base, inference engine, user interface, the SPA software monitor (SPASM), and the SPA mail interpreter (SPAMI). SPASM is responsible for collecting data from the hosts in the network and asserting this data in the appropriate knowledge base objects. SPAMI is responsible for sending mail to users and the system administrator when SPA suspects that problems exist. It is also responsible for receiving mail from users regarding processes that might be involved in performance problems. The two C language programs (HOSTHIST and PROGHIST) are used to collect host and process data at regular intervals even if SPA is not running. This is needed to keep the time series models up-to-date. Currently, HOSTHIST runs once every 10 minutes and collects load average

³A single instance of these is called a *FROB*.

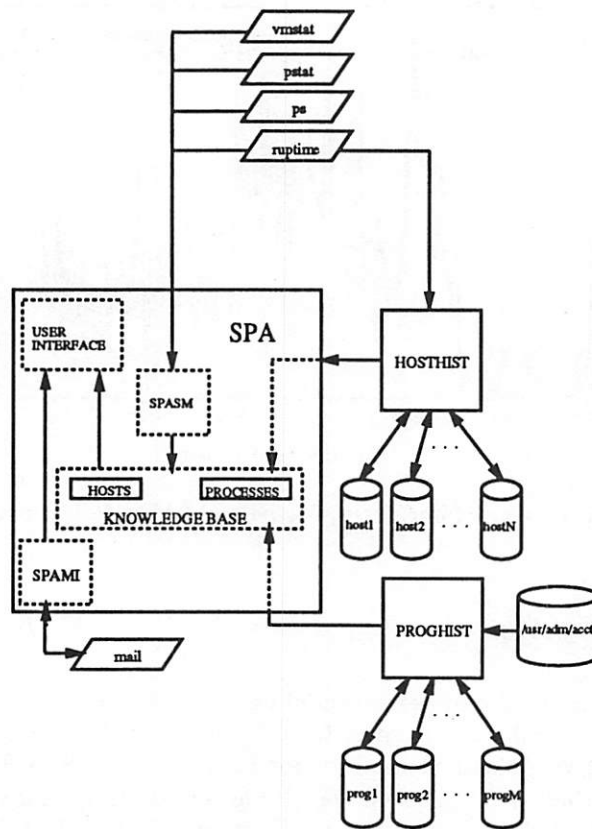


Figure 7: SPA Functional Block Diagram

and user count data from the `rwhod` database. `PROGHIST` is set up to run once every day and collects resource utilization information from system accounting records. `SPA` runs when the system administrator requests it.

The `SPA` system interfaces with the host operating system by invoking standard utilities and a System Administration Tools [15] (SAT) dynamic relation to collect data about hosts and processes in the network. Standard utilities such as `vmstat`, `pstat`, and `ruptime` collect host-level data. The SAT relation (called `kmem`) provides a means to collect process data on any host in the network. Data that are collected by the system are stored in the `SPA` knowledge base. The set of forward-chaining rules, heuristics, and time series models contained in the `SPA` system are used to examine the knowledge base and determine if a problem exists. After a problem is found, another set of rules is used to decide what action should be taken.

For each performance criterion that `SPA` monitors, the time series models used for problem detection are initialized from a history of readings. After the model is initialized, readings are gathered by `SPA` and compared to an expected value determined by the model. Readings that are not within the threshold range result in the creation of an alert that is displayed for the system administrator. The system administrator is responsible for making a determination of the validity of the alert. Validation can be accomplished by examination of the knowledge base, collection of additional data, or by running additional system utilities.

4.1 Problem Detection Within the Network

`SPA` first examines a host's load average to determine whether the host is involved in a performance problem of some kind. To do this, `SPA` retrieves the data maintained by the `rwhod` program and diagnoses one of the problems types listed in Table 1.

Table 1: Description of SPA Host-level Problems

Problem Type	Description
status	the host is down
fileserver_down	a host used as a fileserver is down
load-high	load averages are higher than expected
user-count-high	number of users on the system is higher than expected
wedged-process	1, 5, and 15-minute load averages are identical and non-zero

As described in the previous sections, SPA uses a time series model of a host's load average readings to determine when a load average reading is abnormal. However, the system administrator can specify absolute limits as thresholds for load average problems. Any load average readings over the upper threshold will always cause creation of a problem instance. Likewise, any load average reading under the lower threshold will never cause creation of a problem instance. This approach is similar to the setting of threshold limits that is available today in many commercial system management tools.

The **wedged-process** problem type shows interesting heuristics, and turns out to be very useful in practice. It is diagnosed when there is a host for which all three current load average readings (1, 5, and 15 minute) are the same, non-zero, integral value. This condition is used to detect processes which are hung, or looping in some way on a host that is otherwise little used at the time. This problem type shows up frequently when SPA is first run in the morning: processes that ran overnight and did not complete for some reason will cause the load average to settle in to a non-zero, integral value.

Another interesting rule is **nfsd-processes**. This problem is diagnosed when the load average is "pegged" at the number of **nfsd** processes found. It typically occurs when a naive user runs "find" over our global "root" directory.

4.2 Problem Detection Within a Host

After a host is discovered with a workload that exceeds the threshold value, SPA applies tools with finer resolution and more discrimination, such as the **pstat** and **vmstat** commands and the **kmem** SAT dynamic relation described above. The data about the processes involved in the current workload is compared to per-command profiles gleaned from process accounting records. Again, if there is a process whose data values are found to exceed expected ranges, that process is treated as a source of the performance problem on that host. At the process level, SPA will detect the problem types described in Table 2.

4.3 Interacting With SPA

The system administrator has commands available to inspect the knowledge base, perform actions on the knowledge base and the hosts and processes in the network, and to query SPA about its operation.

Inspecting the Knowledge Base

The **show**, **select**, and **plot** commands are the primary means by which the system administrator inspects the knowledge base. The **show** command takes any number of knowledge base object names as arguments and displays the contents of those knowledge base objects.

The limitations of this approach to inspecting the knowledge base were quickly made evident when SPA was run on a large number of hosts. When there are thousands of knowledge base objects, frequently the system administrator does not know the names of the desired objects, only the qualities of such objects. Thus, a **select** command was implemented which allows querying the knowledge base in the fashion of a relational database.

Table 2: Description of SPA Process-level Problems

Alert Type	Description
<code>kernel_table</code>	one or more kernel tables is approaching 100% used
<code>mem_size</code>	the memory size of a process is larger than expected
<code>time_used</code>	the CPU time used by a process is larger than expected
<code>elapsed_time</code>	the elapsed (wall) time since a process started is larger than expected
<code>cpu-utilization</code>	process or user is using more CPU resources than expected
<code>mem-utilization</code>	process or user is using more memory resources than expected
<code>nfsd_processes</code>	many nfsd processes with a high rate of CPU utilization
<code>scan_rate_problem</code>	the paging algorithm is scanning at a higher than expected rate
<code>abandoned-process</code>	a process has no parent process
<code>reniced-process</code>	a process has been reniced
<code>zombie_process</code>	a process in its final exit state remains on the system
<code>paging_alert,</code> <code>scan_rate_alert</code>	a host's processes are consuming enough memory to cause paging

The user specifies what kinds of knowledge base objects to search (e.g., `host`, `process`, `problem`), which slots of those objects to display, and a search condition. The search condition specifies a relation that must evaluate to true for each knowledge base object that is a member of the displayed result. The `select` command has the most elaborate syntax and semantics of all the SPA commands: it is implemented as a subset of the SQL [1] relational database query language. A formal semantics of the SPA `select` command is given in [6].

After the `select` command is parsed, a FROBS rule is defined. The specified search condition becomes the premise of the rule. The conclusion of the rule contains a call to an auxiliary function that displays the requested slots of the knowledge base objects that caused the rule to fire. After the rule is defined, the inference engine is allowed to run to determine whether there are any knowledge base objects for which the search condition is true. After the rule has fired for all knowledge base objects, the rule is deleted.

In addition to looking at the current readings in the knowledge base, the system administrator frequently wants to look at the past behavior of a host or process. This can be done with the `show history` command or the `plot` command. The `plot` command uses the `gnuplot` utility to provide a graphical representation of the history of readings.

Performing Actions With the Knowledge Base

The `continue` command is the primary means by which the user initiates the process of data collection, data analysis, and problem detection. Once this command is issued, SPA continues this process until one or more problems are detected via the time series models.

When a problem is detected, the system administrator can enter a therapy phase in which SPA guides the system administrator step-by-step through a procedure script to resolve the problem. This is done with the `advise` command. At each step, the system administrator can have SPA do the step, skip the step, perform the step manually, or explain why SPA is recommending this action. These scripts are written in a language that is a subset of Common Lisp with extensions to run Unix commands, provide explanations, and access the knowledge base.

Querying SPA About Its Operation

SPA provides several facilities to assist the system administrator while using the SPA system. The `help` command provides a short description of commands, data collection types, and global variables in the SPA system. The `why` and `diagnose` commands allow the system administrator to ask SPA to explain a

problem diagnosis. The ability to provide explanations of its reasoning and diagnoses is an essential feature of an expert system. Without it, the system administrator never learns to trust the findings of the expert system. In SPA, explanations come in several forms:

- how the system reached a conclusion,
- a recommendation on what to do to obtain more information about the current problem,
- references to other documentation for further information on this kind of problem, and
- suggested solutions (if any) to resolve the problem command.

4.4 Customizing the Behavior of SPA (Extensibility)

One feature of SPA that is important to its usefulness is that it is *extensible*. If system administrators find that there are additional problems that they would like SPA to detect, a new knowledge base rule can be written. The rule is built by specifying the conditions that must be true when the problem is in effect. A time limit for detection of the new problem type can also be specified. For example,

```
include from process \
where user = 'hoogen' and ppid = 1 and ( mem > 10.5 or cpu >= 50.0 ) \
as big_orphan
```

creates an instance of a problem whenever it finds a process owned by the user *hoogen* that has *init* (*pid* = 1) as its parent and is consuming either more than 10.5% of real memory or at least 50% of the CPU cycles. This 3-line `include` statement is translated into a FROBS forward-chaining rule containing 25 lines of Common Lisp code. This rule is checked during SPA's problem analysis phase to see if the conditions have been satisfied in the knowledge base. If so, an instance of the new problem type (e.g. *big_orphan*) is created and displayed.

5 Results

5.1 Time Series Model Validation

We validated the time series modeling approach on one type of data, the load average. This validation could be done on many other types of data, but load average was the most practical for us to collect. Its excellent results suggest that other measures of resource use would also respond well to time series modeling, but those analyses were not performed.

The time series modeling approach was validated by recording the load average readings for 50 hosts. By type of use, 40 can be classified as workstations, and 10 as servers. In our environment, as in most current large installations, the vast majority of machines are workstations. They are either "desk" machines dedicated to a particular person, or "lab" machines used by different people, but usually only one at a time. For this study, however, we selected a disproportionate number of file servers and general use machines, since we expected, correctly, that they would be more difficult to model accurately.

An assumption underlying this analysis is that the observed loads were mostly "normal," e.g., not the result of looping processes. This was almost certainly true, but if it were not, the effect is that our model will be more accurate than predicted from the analysis. It should not significantly affect the actual values we derived.

The 15-minute load averages were collected every 15 minutes for three weeks, yielding 2100 readings for each host. Each host's records were separately analyzed to gain a "feel" for the data. This consisted of searching for the values of six different variables that would minimize an evaluation function. Three variables are the three smoothing constants. A fourth is related to the standard deviation of the model's forecast *error*: a multiple of that standard deviation is chosen to be the threshold for problem detection.

A fifth is a heuristic, based on the semantics of load average, which modifies the threshold to be at least T greater than the model forecast. T is typically in the range of 0.8. The sixth variable is the value chosen for the simulated problem reading.

The evaluation function is a measure of the false positive reports the actual data would generate (i.e., when a data point was significantly higher than the model's prediction), combined with a measure of false negatives. The latter is estimated at each of the 2100 points by pretending ("injecting") a load L higher than actually recorded, and recording whether the model would have flagged it or not. We tried values of L from 0.5 to 1.0, based on the effect of most real-world problems on load average.

Our results showed that:

- Workstations could be modeled precisely, with near perfect accuracy of problem detection. For $L = 1.0$, both false positive and negative rates were substantially less than 1%.
- Servers, however, were more difficult to model accurately, as shown in Table 3. This stems not from additional variance on the servers, but from the higher average load. Their absolute variance is higher, but the injected problem load (L) is an additive constant, the same value as that for workstations. This stems from our attempting to discover even one "runaway" process. This may be an overly stringent requirement.
- Although not as accurate for servers, the models still work well enough to be very useful.
- For almost all hosts, both workstations and servers, the same values of the smoothing constants are optimal. These are α (0.9), β (0.1), and γ (0.2). A few were 1 or 2/10th's different, but this shows that it's generally not worth the trouble to tune these constants on a per-host basis. Winters [17] says that the response of this type of time series model to changes in α , β , and γ is "flat": that is, small changes in the parameters have small effects on the quality of the forecasts. The analysis of the models tested here supports that theory. However, large deviations from the optimal values of the constants did result in substantially worse predictions.
- For workstations, a nearly optimal threshold for problem detection was 2.0 standard deviations above the mean forecast error. Modifying the threshold to be at least T greater than the model's prediction did not significantly help, either for workstations or for servers.
- For servers, a tighter range of T (1.5 standard deviations) was required in order to detect a significant proportion of problems. With $L = 1.0$, for the 10 servers, that cutoff yielded an average false positive rate of 1.8% (range 0-3%), and an average false negative rate of 20% (range 0-83%). For the three servers with the highest average load (1.2), $T = 1.0$ gave false positive rates averaging 5.7% and false negative rates averaging 32%.

Discussion

This analysis shows that the models provide excellent forecasts and provide useful problem identification, on the machines in our environment. The biggest weakness is that the average load is so low. However, this reflects the reality of our, and many others, environments. Servers are more difficult to analyze and will require more filtering by an expert or expert system, but typically they are few in number and file servers, at least, have limited access and are well-controlled. By contrast, users of public workstations are often naive, and frequently create problem (looping, etc.) processes which create poor response for the next user. Therefore, even though the average load on workstations is low, the likelihood of problems is significant.

5.2 SPA Expert System

The SPA expert system has undergone limited validation and tuning on a network of 102 workstations and servers. This was done using two methods: (1) the "trial by fire" approach, and (2) problem injection. Results of both of these methods are described in the following paragraphs.

Table 3: Time Series Model Analysis on Server Machines

Host Name	Std. Dev.	Minimum Forecast	False Positive			False Negative		
		Error	Threshold			Threshold		
		(α, β, γ)	1.0	1.5	2.0	1.0	1.5	2.0
sunset	1.00	0.5, 0.2, 0.1	7	2	1	59	83	92
cs	0.74	0.9, 0.1, 0.2	1	0	0	14	62	96
gr	0.59	0.9, 0.1, 0.2	9	3	1	24	38	68
asylum	0.31	0.9, 0.1, 0.2	8	3	1	4	8	12
jensen	0.29	0.8, 0.1, 0.2	3	0	0	1	2	4
vlsi	0.26	0.9, 0.1, 0.2	4	2	1	3	5	8
jaguar	0.23	0.9, 0.1, 0.2	8	3	0	0	2	4
peruvian	0.20	0.9, 0.1, 0.2	9	3	1	0	0	1
kayenta	0.19	0.8, 0.1, 0.2	4	2	1	0	1	2
ursa0	0.06	0.9, 0.1, 0.2	1	0	0	0	0	0

The "Trial By Fire" Approach

In order to keep performance records, SPA records every time an instance of a problem is created. It also records the number of these problems that the system administrator classifies as valid and invalid. The following is a transcript of the output of the SPA `status` command. The `status` command is used to display the statistics that SPA maintains on problem detection:

```
SPA(H=3,int)> status
PROBLEM STATUS:
LOAD-HIGH:      7 valid /   14 found =   50.0 % success rate.
STATUS:         8 valid /    9 found =   88.9 % success rate.
FILESERVER_DOWN: 2 valid /    2 found = 100.00 % success rate.
KERNEL_TABLE:   2 valid /    3 found =   66.7 % success rate.
MEM_SIZE:        2 valid /    4 found =   50.0 % success rate.
TIME_USED:       1 valid /    1 found = 100.00 % success rate.
ELAPSED_TIME:    0 valid /    1 found =    0.0 % success rate.
MEM_UTILIZATION: 3 valid /    5 found =   60.0 % success rate.
CPU_UTILIZATION: 3 valid /    4 found =   75.0 % success rate.
NFSD_PROCESSES:  0 valid /    0 found =    0.0 % success rate.
SCAN_RATE_PROBLEM: 5 valid /    6 found =   83.3 % success rate.
USER-COUNT-HIGH: 4 valid /    9 found =   44.4 % success rate.
WEDGED-PROCESS:  1 valid /    3 found =   33.3 % success rate.
ABANDONED-PROCESS: 11 valid /   15 found =   73.3 % success rate.
RENICED-PROCESS:  5 valid /    5 found = 100.00 % success rate.
ZOMBIE_PROCESS:  3 valid /    3 found = 100.00 % success rate.
SCAN_RATE_ALERT:  6 valid /    8 found =   75.0 % success rate.
PAGING_ALERT:    6 valid /    7 found =   85.7 % success rate.
OTHER:           0 valid /    0 found =    0.0 % success rate.
TOTAL:           69 valid /   99 found =   69.7 % success rate.

SPA(H=3,int)>
```

The biggest problem with using the "trial by fire" approach to validation of SPA is that it is a slow process. Bugs that are found have to be reported, fixed, and tested. Problems that SPA finds have to be investigated to determine their validity and then resolved. Some problem types occur frequently enough that it is possible to validate SPA's response when the problem occurs. Other problems occur so infrequently that it is impossible to thoroughly test SPA's response without some kind of problem injection.

In general, false positive results were too high for many problems such as memory utilization, for which we did not develop time series models. Either more tuning is necessary to better filter these, or, preferably, analyses could be run to develop time series models for these types of data.

Problem Injection

Problem injection was accomplished by introducing poorly-behaving programs into the environment, and auditing SPA to determine whether it correctly identified them. In general, it did very well for most problems which affected the load average, which is its screening device. `Wedged-process` and `load-high` were all detected at a high rate. `zombie-process`, `abandoned-process`, and `reniced-process` were detected whenever SPA's attention focused on a the host with a load average problem. `Elapsed time` worked well, when the rule was tuned to eliminate certain programs which users tend to keep running. Some other rules did not work as well. The main problem stems from SPA's primitive data acquisition facilities, that present too high an overhead to run frequently, on all hosts. This is an issue outside the domain of expert systems or the time series models, and is being addressed externally, through the SNMP Host MIB development.

6 Related Work

The work most similar to ours was done in the the Intrusion Detection Expert System[10] (IDES) and NIDX[2], computer security monitoring systems. They are based on an intrusion detection model described by Denning [3] for detecting security threats in computer systems. Denning's model is based on the hypothesis that a security violation can be detected by monitoring usage patterns of the system's users. A potential security violation would make itself evident as an abnormal pattern of usage, which is determined by comparing against predictions from time series models. The same idea is used by SPA to detect problems in hosts and processes.

TIMM/Tuner[16] is a computer system performance tuner developed in TIMM, a commercially available expert system shell. Its users are VMS system managers wishing to evaluate the system performance of a VAX computer system running VMS. The system attempts to isolate performance bottlenecks based on information supplied by the user in a question/answer session. The system recommends adjustments to system parameters, or if needed recommends hardware or software upgrades. Unlike SPA, TIMM/Tuner does not collect data in real-time. Although the problem domain is similar to that of SPA, it is a narrower domain because its expertise is limited to DEC computer systems running VMS.

Hitson described an expert system for monitoring and diagnosing problems in TCP/IP-based networks[5]. He concentrates on developing heuristics for diagnosing the ultimate problem, and does not use adaptive models to determine problem thresholds.

Network management systems such as HP's OpenView and Sun's SunNetmanager are an important related area. There are other commercial systems such as Tivoli's, Cabletron's Spectrum, and the forthcoming DME from OSF. Most of these systems are simply frameworks for other (sometimes non-existent) tools, which do the actual work. In the absence of automated tools, the management systems provide a framework into which the system manager inserts simple, limited relations. As far as we can determine, none of these systems even dynamically determines threshold values, although most can deal with proportional measures. They lack the capability of adapting to changes in the data. Typically, the network administrator spends a considerable amount of time determining appropriate threshold values for the data monitored in the network. If the usage of the network changes significantly, threshold values may become invalid (causing either a flood of alarms or a complete absence of alarms). Our time series models address this problem by adapting to changes in system usage.

Because current commercial computer and network management tools have limited abilities to combine criteria in complex ways, they require extensive per-host configuration to detect particular problems. SPA provides much more power and flexibility with its support of an SQL subset, its ability to be extended at run time, and its powerful and extensible knowledge base facts and rules.

Standardized SNMP Host MIBs that enable better, faster, and cheaper access to measurements of host state are an important step in the development of better diagnostic and management tools.

7 Future Work

SPA is a useful prototype to validate our time series model and the expert system itself. However, as a large Lisp system, it is undesirably costly in processor cycles and memory use. Portions of it could be re-implemented in C++. The use of standard (and costly) Unix utilities to gather raw data could be replaced by a custom daemon. (When SNMP host MIB's are widely available, those can be used.) With these efficiency improvements, SPA could prove of significant practical use.

There are many additional problem areas SPA might diagnose, such as the network itself, disk utilization, and system security. One of the problems in monitoring all the hosts in a large network is the volume of problems, alerts, and messages that gets generated. A more effective means of filtering this information could be provided, or more aggressive use of time series models could be used.

Additional extensibility of the SPA system could be provided. The `include` command provides a primitive form of extensibility by allowing a means by which the system administrator can define new problem types. However, frequently, it is useful for the system administrator to define procedural actions to take when specific conditions occur. By allowing the system administrator to specify an arbitrary set of actions in the `include` command, powerful extensibility can be provided.

8 Conclusion

A time series model is an effective, practical, easy to implement technique for determining problem threshold values in computer systems. It alleviates one of the most significant weaknesses of current computer and network management systems: the manual determination of a fixed threshold value. The effectiveness of the time series model is being demonstrated by its use in the SPA system administration expert system. The sophistication and power of an expert system is appropriate, and we suspect required, for effective system management in today's large computer networks.

Acknowledgements

We are grateful to Sean O'Neill for analyzing the times series models. Our thanks for reviewing earlier drafts of this paper go to Robert Kessler, Mike Hibler, and Sean.

References

- [1] American National Standards Institute, New York, NY. *X3.135-1989: Database Language — SQL with Integrity Enhancement*, 1989.
- [2] David S. Bauer and Michael E. Koblenz. NIDX — a real-time intrusion detection expert system. In *Proceedings of the Summer 1988 USENIX Conference*, pages 261–273, 1988.
- [3] Dorothy E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.
- [4] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
- [5] Bruce L. Hitson. Knowledge-based monitoring and control: an approach to understanding the behavior of TCP/IP network protocols. *Proc. of the SIGCOMM '88 Symposium on Communication Architectures and Protocols*, 18(4):210–221, August 1988.
- [6] Peter James Hoogenboom. Semantic definition of a subset of the structured query language (SQL). Technical Report UUCS-91-026, University of Utah, December 1991.

- [7] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.
- [8] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, MA, 1989.
- [9] Colin D. Lewis. *Industrial and Business Forecasting Methods*. Butterworth Scientific, London, 1982.
- [10] Teresa F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 59-66. IEEE Computer Society Press, 1988.
- [11] Douglas C. Montgomery, Lynwood A. Johnson, and John S. Gardiner. *Forecasting and Time Series Analysis*. McGraw-Hill, Inc., New York, NY, 2nd edition, 1990.
- [12] Eric George Muehle. FROBS: a merger of two knowledge representation paradigms. Master's thesis, University of Utah, 1987.
- [13] J. H. Saltzer and J. W. Gintell. The instrumentation of MULTICS. *Communications of the ACM*, 13(8):495-500, 1970.
- [14] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 2nd edition, 1988.
- [15] Brian Sturgill. System administration tools. Master's thesis, University of Kentucky, 1989.
- [16] Donald A. Waterman. *A Guide to Expert Systems*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [17] Peter R. Winters. Forecasting sales by exponentially weighted moving averages. *Management Science*, 6(3):324-342, April 1960.

Author Information

Peter Hoogenboom recently received his Masters degree at the University of Utah. The SPA expert system was the subject of his Masters thesis. In his eight years of experience in the industry, Peter has worked in a variety of capacities, including the design and analysis of real-time simulation systems, system administration, and expert systems for UNIX system administration. His current projects include porting GNU tools to the HP PA and the development of the OMOS Object File Editor (OFE). Since graduating with his Masters degree, Peter has become a full-time staff member in the Center for Software Science.

Jay Lepreau is Assistant Director of the Center for Software Science, a research group within Utah's Computer Science Department which works in many aspects of systems software. He has worked with Unix since 1979, and has served as co-chair of the 1984 USENIX conference and on numerous other USENIX program committees. His group has made significant contributions to the BSD and GNU software distributions. His current research interests include dynamic software system structuring for performance and flexibility, with operating system, language, linking, and runtime components.

The author's addresses are: Center for Software Science, Department of Computer Science, University of Utah, 84112. They can be reached electronically at {hoogen,lepreau}@cs.utah.edu.

Design and Implementation of a Simulation Library using Lightweight Processes*

Janche Sang
Ke-hsiung Chung
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

The *St* lightweight-process based system for simulating process interactions is an enhancement to the C programming language in the form of library primitives with sets of predefined data structures. The *St* system encapsulates an existing lightweight-process library to provide a discrete-event simulation environment supporting the process view. It was developed as a research testbed for investigating methods which support simulations efficiently. Easy extensions and modifications to the *St* system are a major design objective, accomplished through modularity and layering. This paper describes the system, our experiences with its implementation, and its applicability to simulation modeling. We report on performance measurements of different implementations of the simulation scheduler, and of different algorithms for simulating service disciplines.

1 Introduction

The process view of simulation, developed and refined over the past two decades [6, 8], is a view that enables an analyst to model a discrete-event system in terms of interacting processes. A key advantage of the process-interaction approach to simulation modeling is that model description via processes and their interactions makes the entire modeling activity, from model design to code debugging and execution, require relatively little effort. This is in comparison to the level of effort required in alternate techniques, such as the event-scheduling simulation method, which offers a lower level of abstraction than the process-interaction method.

In the process view of simulation, processes employ a variety of statements to define the flow of entities (transactions, customers, jobs, etc.) through the system. Two or more processes may compete for resources and exchange messages with one another for the purpose of synchronization. The relatively high level of abstraction enjoyed by process-oriented models make model construction relatively effortless. Another benefit is the added flexibility given by application-level processes, and the ease with which process-based models can be scaled to implement simulations of large systems [12]. In this work we describe the design of a process-interaction system based on lightweight processes. *St*, a system for Simulating process *i*nteractions, is a process-oriented discrete-event simulation system. It has been designed to enhance the capabilities of the C programming language through a set of primitives which provide a quasi-parallel programming environment.

The primary goal of this work is to develop a research vehicle for conducting experiments and obtaining measurements in empirical evaluations of algorithms used in simulation systems. Such a research vehicle helps in identifying principles of good design and alternatives which contribute toward efficient, accurate, and reliable simulation software. A secondary goal is to present an interface that is simple and straightforward, but sophisticated enough to meet the needs of a wide range of applications. The primitives provided in the *St* skeleton are sufficient for a variety of tasks, including process manipulation and synchronization, random number generation, and statistics collection - all of which enable the description of an application to the maximum extent possible without application-specific details.

There are two main approaches to designing process-oriented simulation software. One approach is to de-

* Research supported in part by NSF award CCR-9102331, NATO-900108, ONR-93-1-0233, and ARO-93-G-0045.

sign a specific simulation language. Several existing languages such as SIMULA[1], GPSS[10], etc. belong to this category. The other approach is to construct and place simulation primitives on top of an existing language. Examples of this approach can be found in [2, 11, 20, 22], where Ada, Extended Pascal with coroutines, C, and Modula II have been used as target languages. We decided to take the second approach with *S_i* for the following reasons. First, it requires less effort to extend and modify a library than to design and implement a new language. Second, users tend to be more comfortable with using a familiar and trusted language instead of having to learn yet another new language [7].

To construct an efficient process-oriented environment, we used so-called lightweight processes which can operate within a single address space. In fundamental structure, a lightweight process is no different from a process; each has its own stack, local variables, and program counter. However, as compared to a process, a lightweight process is lighter in terms of overheads associated with creation, context-switching, interprocess communication, and other routine functions. We decided to base *S_i* on the Sun Lightweight-Process library (LWP) [24]. We chose this as a kernel because of its availability in our research environment, its capacity for reliable context-switching and allocation of protected stacks, and its continued development. Further, ease of portability of the system, due to potential conformation with POSIX standards makes the library attractive.

The remainder of the paper is organized as follows. In Section 2 we briefly introduce the Sun Lightweight-Process library. Some design issues are discussed in Section 3, and Section 4 details the implementation of the four major modules in *S_i*. Section 4 contains a description of the process management, process coordination, resource management and statistics functions. In Section 5 we illustrate some of the strengths of the *S_i* system, with early measures of performance given in Section 5.1 and 5.2, a simple but remarkable performance enhancement, easily implementable in *S_i*, described in Section 5.3, and a brief comparison of different algorithms for simulating service disciplines given in Section 5.4. A short conclusion is presented in Section 6.

2 Design Issues

The design of the *S_i* system was motivated by three primary objectives, namely,

- to provide a simple and effective interface which encapsulates the LWP library,
- to provide modularity that supports easy extensions and modifications, and
- to achieve efficient simulation executions.

The *S_i* system employs the LWP library as its kernel. A layered design enables applications to be created without direct utilization of LWP functions. Therefore, any changes to the LWP library is transparent to the application level. *S_i* was designed with a modular structure to simplify the replacement and modification of specific parts of the simulator. Modularity is well-recognized as an important concept in system design because it provides for convenient system maintenance. This property is invaluable in software systems like *S_i* which are used as research testbeds, frequently undergoing modifications to improve existing algorithms or to add new features which improve system functions and performance. For example, two different approaches are provided in *S_i* to simulate the round-robin service discipline. The modular structure allow us to implement these two algorithms without modifying system structure.

The performance of executing simulations is a major concern in our design. Since simulations are usually time-consuming, the need for significantly reducing the execution time of simulation programs is a critical one. One source of overhead which exhibits the potential to contribute to long execution times is due to context-switching between lightweight processes. Although a lightweight process's context-switching overhead is cheaper than that of its heavier UNIX counterpart, it nevertheless can still contribute in a significant way towards execution time, particularly when the number of context-switches is large. Another source of overhead may be attributed to *S_i*'s layered design, which results in a series of function calls. This is unavoidable, unless the layers are broken up and combined. But the overhead can be minimized if only a thin layering of sparse code is used in *S_i* to encapsulate LWP primitives.

In general, our objectives in designing and building the *S_i* system are to build a layered, modular, and efficient simulation system for process-interaction based simulations. Our eventual goal is the integration of the *S_i* and EclIPSe systems [15, 25]. The latter system provides high level simulation primitives which enable simulation tasks to be executed in parallel. With such support, the applications developed in *S_i* can be conveniently executed on networks of heterogeneous workstations. The requisite layering for such a system is shown in Figure 1.

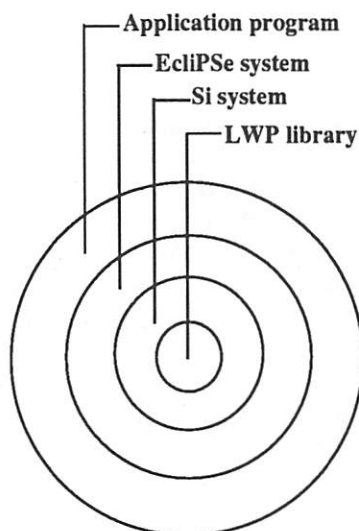


Figure 1: System Layers

3 Implementation Issues

The *Si* system consists of four major components, namely, a process management component, a process coordination component, a resource management component, and a probability and statistics library. Figure 2 depicts layout of components and examples of functions provided within each.

3.1 Module Descriptions

The process creation task in *Si* is achieved through an invocation of the function *si_create(func, nargs, arg1, ..., argn)* which encapsulates the LWP function *lwp_create()*. Following the creation of a process, another function invocation *si_insert(E)* is used to ensure that the specified process with reactivation record *E* is reactivated at time *E.clock* by saving the record in the simulation calendar.

Process Management

Transfer of control between specific processes is most easily done with the aid of the *lwp_yield()* function. However, a problem arises because the LWP library provides its own process scheduler which schedules processes based on LWP scheduling priority; the scheduling rule always selects a process with the highest priority to run. If more than one such process exists, the scheduler uses a FIFO policy within the priority class. In building *Si* on top of the LWP library, a potential problem arises when an executing process terminates. At this point, the LWP scheduler will transfer control to the highest LWP priority process, one which is not

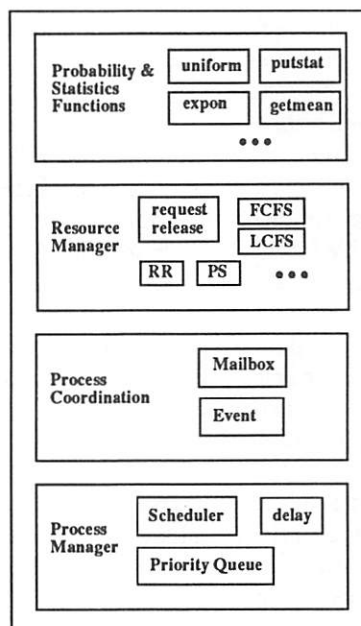


Figure 2: The modules in *Si*

necessarily the process whose execution is imminent in simulation logic, i.e., one whose activation record in the simulation calendar has the smallest reactivation time. While an apparent solution is to require a mapping between priorities in *Si* and priorities in LWP, the integer priority formats used in LWP preclude the use of this option because *Si*'s priorities are double precision numbers, representing reactivation (simulation) times. Therefore it is necessary to determine an alternate scheme for transferring control between a terminating process and *Si*'s scheduler.

Our strategy is to force the LWP scheduler to schedule only a limited number of processes in *Si*. To achieve this, each process is assigned an LWP priority, either a MINPRIO (minimum priority) value or a MAXPRIO (maximum priority) value. This principle is stated as follows:

Principle: At any given instant, at most two processes exist with the highest priority value MAXPRIO. One of these is *Si*'s process scheduler, and the other is a currently executing application-level process.

With this principle, when a currently executing process terminates, the process scheduler obtains control under the LWP library's own scheduling policy since it is the unique process left with priority value MAXPRIO. This solves the control transfer problem. Figure 3 shows the pseudo-code required for the process scheduler. As implied by the principle, an executing

process scheduler selects the highest priority process, say process *pid*, from the simulation calendar, using smallest reactivation time as a measure of priority. It raises the LWP priority of process *pid* to MAXPRIO and subsequently transfers control to process *pid*. When the process scheduler is invoked due to process suspension, the scheduler follows exactly the opposite routine. It reduces the LWP priority of the suspended process, say process *pid*, from a value MAXPRIO to a value MINPRIO, prior to saving its reactivation record in the simulation calendar, if necessary. Observe that two distinct processes are created within the *main()* routine shown in Figure 3. The first process executes a *si()* function which the application code treats as a main routine. The second process is the process scheduler whose function it is to execute the *schedule()* routine. Both are created with the highest priority value MAXPRIO. The notation $\mathcal{S}i_p$ is used to emphasize the fact that the scheduler is implemented as a process here.

An alternate approach toward implementing transfer of control between processes is through function invocations. To demonstrate this point, we implement a different version of the process scheduler, through a small modification of the function *schedule()*. In the remainder of the text, we use the notation $\mathcal{S}i_f$ to emphasize the use of function invocations in the implementation of the scheduler. The function *schedule()* in $\mathcal{S}i_f$ uses a strategy similar to that used in $\mathcal{S}i_p$ to switch control between processes. Since the scheduler in $\mathcal{S}i_f$ is no longer a process, an executing application-level process is the only process with the highest priority value MAXPRIO. First, in function *schedule()*, the selection of the next process to execute, say *pid*, is made by accessing the simulation calendar. Following this, process *pid*'s priority is raised to value MAXPRIO, and the currently executing process (i.e., the process seeking suspension) reduces its own priority by invoking function *lwp_setpri()*. Through this priority reduction scheme, control is transferred to the single remaining highest priority process, i.e., *pid*.

The scheme described above allows for transfer of control between a process requiring suspension and a process whose execution is imminent. Unfortunately, a problem arises when an executing process terminates naturally, without either voluntarily or involuntarily requesting suspension. Upon its termination, there is no unique highest-priority process to take control. Left to its own devices the LWP scheduler would give control to the first process in line within the single priority class shared by all other processes. Needless to say, this would result in incorrect simulation logic. To get around this problem, we use an additional function called *si_exit()* which a process must invoke just prior

to its termination. The function *si_exit()* invokes function *schedule()* which uses function *lwp_destroy()* to eliminate the currently executing process, effectively a suicide operation. As before, *schedule()* also selects a process whose execution is imminent, so that the invocation of function *lwp_destroy()* causes control to switch to the new high-priority process. These actions are summarized in the pseudo-code shown in Figure 4.

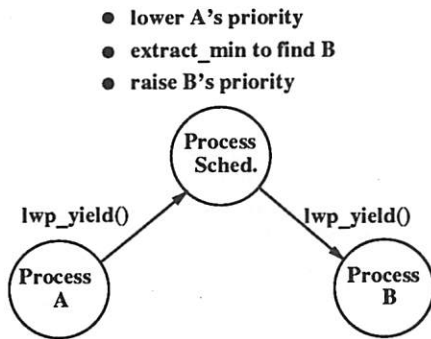
It is worth mentioning that there is yet another way to solve the "natural termination" problem. By modifying the return address of a process, control can be transferred to some function specified in the address when the process terminates naturally. This can be accomplished either by modifying *lwp_create()* to set the address of the function *schedule()* as the return address, or by implementing a new thread creation routine which achieves the same effect [9]. Since our design goals emphasize high level abstractions and layered design, we chose not to adopt this option.

There is another important function in $\mathcal{S}i$ known as the *delay()* function. When an executing process decides to suspend itself for *t* units of simulated time, it invokes the function *delay(t)*. This function inserts the invoking process's reactivation record, including its reactivation instant *clock + t*, into the simulation calendar. Since the invoking process must undergo suspension, the process scheduler selects as the next process to execute that process in the simulation calendar with the lowest reactivation time. In the $\mathcal{S}i_p$ design, transfer of control to the process scheduler is done via the *lwp_yield(scheduler)* action, while in the $\mathcal{S}i_f$ design, the scheduler is invoked directly through a call to function *schedule()*. This is described in the pseudo-code shown in Figure 5.

Process Coordination

The $\mathcal{S}i$ system provides two distinct coordination mechanisms to support synchronization between processes. One mechanism is through user-declared events, effected by calling *wait_event()* and *set_event()* primitives. A process is suspended if it invokes function *wait_event(e)* because it is forced to wait until event *e* occurs. Event *e* is said to occur when some other process invokes function *set_event(e)*. At this point, all processes waiting for event *e* are reactivated simultaneously. In $\mathcal{S}i$, an event *e* is declared to be of type *Event* and initialized by the *create_event()* function.

The other mechanism for process synchronization is through message-passing. There is a predefined data structure called *Mailbox* which is created by function *create_mailbox*. Messages can be sent and received through the mailbox by using functions *si_send()* and *si_receive()*, respectively. The function *send(mb,msg)*

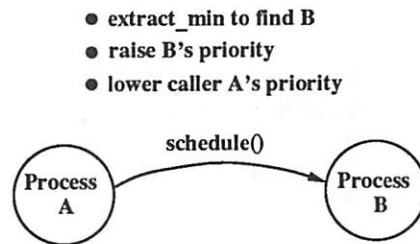


```

schedule()
{
    while(future_event_set is not empty) {
        if previously executing process is
            suspended
            lwp_setpri(ppid,MINPRIO);
        E = extract_min(future_event_set);
        clock = E.clock;
        lwp_setpri(E.pid,MAXPRIO);
        /* Now only E.pid and the scheduler
           have highest priority.*/
        si_yield(E.pid) /* resume execution of
            process pid */
    }
}

main()
{
    int si();
    ...initialization...
    lwp_create(si_tid,si,MAXPRIO,...);
    lwp_create(sched_tid,schedule, MAXPRIO,
        ...);
}
  
```

Figure 3: Process Scheduling in Si_p



```

schedule()
{
    while(future_event_set is not empty) {
        E = extract_min(future_event_set);
        clock = E.clock;
        lwp_setpri(E.pid,MAXPRIO);
        if the current process is dying
            lwp_destroy(cpid); /* suicide*/
            /* control transfers to E.pid
               automatically. */
        else
            lwp_setpri(cpid,MINPRIO);
            /* control transfers to E.pid
               automatically. */
    }
}

main()
{
    int si();
    ...initialization...
    lwp_create(si_tid,si,MAXPRIO,...);
}
  
```

Figure 4: Process Scheduling in Si_f


```

delay(t)
{
    E.pid = CurrentPID;
    E.clock = clock + t;
    si_insert(E);
    lwp_yield(scheduler);
    /* or schedule() in Si_f */
}

```

Figure 5: The function *delay()*

deposits the message *msg* in the mailbox *mb*. If there is a process awaiting the arrival of this message, *si_send()* enables the process to access the message and consequently be reactivated. The reverse function *si_receive(mb,&msg)* obtains the message from the mailbox. If no message is available, the invoking process has to be suspended until a message arrives. For simplicity, the size of a message is limited to one word (i.e., the size of an integer or pointer). This is patterned after the design of the XINU system [3].

Resource Management

In contrast to processes which are used to model active components of a system, resources are used to model passive system objects with mutually exclusive access. In other words, processes request access to resources, use these resources for a certain length of time, and finally release them and proceed with different activities. The *Si* system supports two basic functions for resource access, called the *request(r)* and *release(r)* functions (see Figure 6). The resource object is declared to be of type *Resource* and initialized by a *create_resource()* function. When a resource *r* is occupied, other requesting processes must wait in a queue associated with resource *r*. When resource *r* is released, a suspended process in the front of the queue is given permission to resume, with access to *r*.

In a real application, a variety of queueing disciplines is possible, including first-in first-out (FIFO), round-robin (RR), processor-sharing (PS), etc.. Some, such as the latter two, are more complicated than others. These complex disciplines utilize different rules in selecting the next process to execute, when faced with a choice. To give analysts a common interface to these disciplines, the *Si* system employs a function *use(r,t)* (patterned after CSIM [20]) to allow a process to utilize a resource *r* for a given length of time *t*. The queueing discipline is specified as a parameter when resource *r* is initialized. For example, the statement

```
r = create_resource(ps)
```

binds resource *r* with the *ps* function which is prede-

fined in *Si* to simulate the PS discipline. This simple interface also provides a level of modularity which lets analysts develop their own queueing discipline in a fairly effortless manner.

Probability and Statistics Functions

The *Si* system provides some necessary random number generation functions such as *uniform()* for generating deviates from a uniform distribution, and *expon(u)* for generating deviates from an exponential distribution with mean *u*, etc.. In addition, *Si* supports two types of statistics. The first type is a predefined type, involving data that is implicitly associated with a resource to be automatically collected, summarized and reported. For example, applications can use functions such as *util(r)* and *qlen(r)* to obtain the utilization of and queue-size at resource *r*. The second type is a user-defined type, requiring user-tables for statistics collection. For example, an explicit invocation of the function *putstat(t,x)* inserts datum *x* into a user-defined table *t* which is declared with the type *Table*. The sample mean and variance can be obtained by calls to functions *getmean(t)* and *getvar(t)*, respectively.

At present, the *Si* system also supports functions *reset_resource(r)* and *reset_table(t)* to clear the statistic-collection fields in resource *r* and table *t*, respectively. There are two advantages to using these functions. The first advantage is that these functions can be used to eliminate the effects of the start-up transient in simulations. The second is that they can be used in the regenerative simulation method, where independent samples are obtained from independent cycles of a simulated system [4].

4 Performance Measurements

In this section we first present some performance measurements obtained from the *Si* system, a simple performance enhancement which can significantly reduce simulation execution time, and finally some specific algorithms for service disciplines. An important reason for pursuing modular design and layering with the *Si* system is potential for easy modification, and thus use of *Si* as a testbed for new ideas in simulation. In the following subsections we present two different examples of how this approach has proven beneficial. The first example is motivated by recognition of the simple fact that simulation execution performance can be significantly enhanced if the scheduler's interaction with the simulation calendar is slightly modified. The second example demonstrates that new, computation-based algorithms for simulating service disciplines more efficiently than direct process-mapped algorithms, are easy to incorpo-

```

request(r)
{
    if (resource r is free)
        flag r as occupied;
    else {
        compute required statistics;
        insert current process's pid at tail of queue.
        lwp_yield(scheduler);
        /* or schedule() in  $Si_f$  */
    }
}

release(r)
{
    if (waiting queue for resource r is not empty) {
        remove process pid from head of queue;
        compute required statistics;
        si_insert(E); /* with E.pid = pid, E.clock =
            clock */
    }
    else {
        compute required statistics;
        flag resource r as free;
    }
}

fcfs(r,t) /* first-come, first-served */
{
    request(r);
    delay(t);
    release(r);
}

ps(r,t) /* processor-sharing */
{
    insert_into_pool(r,t);
    lwp_yield(scheduler);
    /* or schedule() in  $Si_f$  */
    /* regain control from scheduler */
    delete_from_pool(r);
}

...

create_resource(fp)
{
    ...initialization...;
    r->fp = fp; /* fp is a scheduling function pointer
        */
    ...
}

use(r,t)
{
    (*(r->fp))(r,t);
}

```

Figure 6: Resource Management in Si

Operations	LWP	Si_p	Si_f
Creation + Deletion	660	960	860
Process Switches	70	220	170

Table 1: Latency of Operations (in microseconds)

rate in a modular Si system. The configuration used in our measurements was a Sun SPARC IPC workstation (15.7 MIPS, 8 MB memory, 64KB cache) running SunOS 4.1.

The measured times presented here are averages, and it should be emphasized that the measured averages are not intended to represent absolute performance but rather relative performance for a particular parameter configuration. Thus the comparison of average times is of more interest than a comparison of raw numerical data.

4.1 Cost of Operations

In Table 1 is shown a set of measured system overheads for the tasks of process creation and context switching in both the Si_p and Si_f subsystems. Also included, for the purpose of comparison, are the corresponding overheads in the LWP library. Not surprisingly, the Si_f subsystem exhibits less context-switching overhead than its Si_p counterpart, primarily due to its use of a function, instead of an additional process, for process scheduling.

4.2 Benchmark Measurements

A pragmatic approach toward evaluating the performance of the Si system is through the use of benchmark models, using both ease of model development and execution times as indicators of performance. Though our primary interest is in the Si_p and Si_f subsystems, we have included the CSIM system as a basis of comparison. We chose CSIM because it is a sound C-based process-oriented simulation system that has been gaining an increasing amount of popularity in the modeling of complex computer systems [21]. In addition, using the same language (i.e., C) to realize models makes the coding of equivalent definitions considerably easier. In this subsection, we develop models for a single-server queueing system, and a multiqueueing system for a token ring local area network. Both models have previously been used as benchmarks in comparing simulation systems [16, 22].

Benchmark I: A Single Server Model

```

#include <si.h>
#define NMAX 10000 /* # simulated cust. */
#define SM 4.0 /* mean service time */
#define IM 5.0 /* mean cust. interarrival time */
Resource r;
Event done;
si()
{
    r = create_resource();
    done = create_event();
    si_create(gen_cust,0);
    wait_event(done);
    printf("%f, %f\n", qlen(f), util(f));
}
gen_cust()
{ int k;
  for(k=0; ; k++) {
    delay(expon(IM));
    si_create(cust,1,k);
  }
}
cust(k)
{
    request(r);
    delay(expon(SM));
    release(r);
    if(k==NMAX)
        set_event(done);
}

```

Figure 7: A Single Server Queueing Model

The first model is simple, describing a single-server queueing station. The customer arrival process is Poisson, and customer service times are independent, exponentially distributed random variables. We assume that customers are served in their order of arrival (i.e., FIFO discipline). Figure 7 shows the code of this model.

In the first experiment, we execute the benchmark program to measure average execution times versus a varying number of simulated customers. Each execution is repeated several times, using different random number seeds in each case, and the average execution time is computed.

Benchmark II: A Multiple Queue Model

The second model is a little more elaborate, utilizing a multiqueue system with roving server to emulate a token-ring protocol [23]. The multiqueue system represents N independent computer stations situated on a ring, and the roving server represents the token. Messages made up of packets are generated by each station

	Simulated Customers ($\times 100$)				
	10	50	100	200	500
St_p	1.5	7.4	14.8	29.7	73.8
St_f	1.2	6.1	12.1	24.0	60.8
CSIM	1.5	7.6	15.1	30.4	75.9

Table 2: Execution time (in seconds) for Benchmark I

for transmission to other stations on the ring. A single token is passed unidirectionally from one station to its successor on the ring, to provide stations with a mechanism for conflict-free access to the ring for packet transmissions. A station which acquires the token and has queued packets is allowed to complete transmission of a single packet before relinquishing control of the token to the succeeding station on the ring. It is of some interest to determine queueing characteristics of packets at different stations as a function of ring parameters and station traffic.

The parameters of the model include message interarrival time distributions, and packet transmission time distributions at the different stations on the ring. For convenience, we assume that all interarrival time distributions are identical, each being exponential with mean $1/\lambda$. Also, for convenience, assume that all packet transmission time distributions are identical, each being exponential with mean $1/\mu$. Finally, assume that the token-passing time between consecutive stations on the ring is a small constant, a function of ring delay. A similar setup which uses CSIM to model the token ring network can be found in [5].

Empirical Results and Interpretation

The results of both experiments, given in Tables 2 and 3, suggest that the two St subsystems are competitive with CSIM in terms of performance. In particular, the St_f system outperforms CSIM by up to 20%. We emphasize that this does not imply the St system is better than CSIM in all respects. Indeed CSIM is a very stable system, developed and used over several years, while St is still an experimental and evolving system.

It is interesting to observe that the St_f subsystem consistently outperforms the St_p subsystem, even attaining a 40% improvement in execution time for Benchmark II. This is simply due to the fact that St_f employs functions to schedule processes instead of using a dedicated process for this task. Consequently, St_f suffers significantly less context-switching overhead. Based on the empirical results, we can claim that we have achieved a certain level of efficiency, one of the main considerations in our original set of objectives.

	Simulated Customers ($\times 100$)				
	10	50	100	200	500
St_p	3.5	16.7	33.5	67.3	168.5
St_f	2.1	10.0	19.9	39.7	100.3
CSIM	2.6	13.1	26.1	51.9	130.8

Table 3: Execution time (in seconds) for Benchmark II

4.3 A Scheduling Enhancement

Upon examining the code given for the function *delay()* (see Figure 5), it will be clear there is some likelihood that a process reactivation record which has just been inserted into the simulation calendar may very well represent the process whose execution is imminent. In such a case, the insertion of this activation record will immediately be followed by its deletion from the simulation calendar. Clearly, the cost of insertion and subsequent deletion can be avoided if the process in question is recognized to be the process whose execution is imminent. Apart from insertion and deletion savings, unnecessary context-switches between such a process, i.e. one undergoing a potential suspension, and the scheduler may be avoided. Recognition of such a situation entails a comparison operation in which the scheduler determines if the simulation priority of the process in hand is greater than the simulation priority of the highest priority process in the simulation calendar. Because this comparison operation must now be done for each process scheduled for execution, there is a trade-off between the new scheme (in terms of the additional comparison cost) and the old scheme (where there is no comparison cost, but avoidable insertion-deletion pairs of operations and context-switches).

In order to obtain a rough assessment of the frequency of such unnecessary insertion-deletion actions of process reactivation records, we measure the ratio of such occurrences to the total number of times that function *delay()* is invoked. Recall that invocation of function *delay()* initiates a process's suspension by a control-switch from the process to the scheduler, and a subsequent control-switch to a new, or the same, process. Table 4 contains percentages of such unnecessary actions for both Benchmark programs. Surprisingly, the avoidable cost can be seen to be as high as 78% for the multiqueue model.

This simple idea is incorporated in St_i through a small modification of the original code in the function *delay(t)*. An additional function, *findmin()*, is required for actually performing the comparison. It determines if the highest priority process in the simulation calen-

```

delay(t)
{
    E = findmin(future_event_set);
    if( t+clock < E.clock ) {
        /* no need to insert; process continues
           to execute */
        clock += t;
        return;
    }
    else {
        ... original code in delay() ...
    }
}

```

Figure 8: Add an extra check to *delay()*

dar, with reactivation record *E* and reactivation time *E.clock* is to be given control before or after the function that invoked *delay()* and requires control at simulation time *clock + t*. Clearly, if the quantity *E.clock* is smaller, then the currently executing process must be suspended; otherwise, the savings will include an insertion, a deletion, and two context-switching actions in the case of St_p . In the latter situation, the process continues execution, upon an immediate return from function *delay()*. A succinct description of this scheme is given by the pseudo-code shown in Figure 8.

Using the modified *delay(t)* function, we repeat the experiments described above to obtain average execution times for the two benchmark models. The results, shown in Table 5, indicate a significant improvement in performance as compared to the previous results (see Tables 2 and 3). The impact of this modified piece of code on the performance of the St_p subsystem is larger than its impact on the St_f subsystem. This is largely attributable to the fact that both, process switching overheads and simulation calendar overheads are reduced in St_p , while only simulation calendar overheads are reduced in St_f . Though the difference in performance between the St_p and St_f subsystems decreases with the comparison modification, the St_f subsystem is still a consistently better performer than the St_p subsystem.

The performance improvement to be had from the additional comparison operation depends on the frequency of the desired property (i.e., the currently executing process must continue execution without incurring calendar and context-switching overheads) relative to the actual cost of performing the comparison for every process that invokes the *delay()* function. It should be apparent that the more frequent the condition supporting the property, the larger will be the savings. Also, the larger the ratio of this count to the number

	<i>saved/total</i>
Benchmark I	27802/100000 \approx 28%
Benchmark II	337954/433437 \approx 78%

Table 4: Percentages of the saved Insert and Extract_min operations

		Simulated Customers ($\times 100$)			
		100	200	500	gain
Benchmark I	St_p	13.6	27.2	67.9	8%
	St_f	11.8	23.5	59.2	2%
Benchmark II	St_p	18.9	37.6	94.2	44%
	St_f	16.9	34.0	84.7	16%

Table 5: Execution time (in seconds) for Model I and Model II (improved)

of times the *delay* function is invoked, the more gain is to be had by adding this comparison test. Using f to denote the frequency with which the condition is true, C_{test} and $C_{overhead}$ to represent the costs of comparison and overhead, respectively, and r the ratio $C_{test}/C_{overhead}$, the enhancement is beneficial whenever

$$C_{overhead} > C_{test} + (1 - f) \times C_{overhead} \quad (1)$$

or equivalently, whenever

$$f > \frac{C_{test}}{C_{overhead}} = r. \quad (2)$$

In support of this modification, it is known that when scheduling distributions (which dictate how reactivation-times are dispersed in the simulation calendar) are mixture distributions, reactivation-times tend to pile up towards the beginning of the simulation calendar [13]. Because such scheduling distributions are realistic, the modified scheme is likely to almost certainly yield reduced execution times for most applications. A detailed analysis of the savings given by this method can be found in [17].

4.4 Round-Robin and Processor-Sharing Algorithms

Service disciplines such as first-in-first-out (FIFO), round-robin (RR), processor-sharing (PS), etc. are functional parameters of service stations in queue-based simulations. As systems that are designed and built become increasingly complicated both in functionality

and description, we are faced with a choice between hypothetically weak (in the sense of model assumptions) analytic models and conclusively weak (in the sense of execution data) simulation models. In most instances of practical interest we usually have little choice but to rely on good simulation models to answer questions related to system performance. Hence efficient techniques for implementing simulation algorithms, including algorithms for service disciplines, are useful as simulation execution enhancements.

In the round-robin (RR) service discipline, a job is serviced for a single quantum q at a time, sharing the service resource with other jobs undergoing the same service allocation. If the remaining service time required by a job exceeds the quantum size q , the job's processing is interrupted at the end of its quantum and it is returned to the rear of the queue, awaiting the service quantum it will receive in the next round. Instead of taking a naive approach which simulates the round-robin discipline by physically switching control from one process to another, we propose a computational scheme which is based on predicting departure instants of serviced jobs leaving the pool of queued jobs. The computed departure instant of a particular job may be invalidated by one or more newly arriving jobs, i.e., one or more arriving after the corresponding departure event is scheduled, but before the departure event can occur. This is because the server must now attend to one or more previously unaccounted for job arrivals, and decrease the amount of attention it planned on giving to jobs already in the system prior to the arrival of the new job(s). Consequently, a scheduled departure event that has been invalidated in this manner must be cancelled, and an updated departure instant for the same or another job scheduled in its place. In addition, it is necessary that the remaining service time requirements of each job in the system be adjusted whenever an arrival event or a departure event occurs.

The processor-sharing (PS) discipline schedules jobs as if the server were processing all the jobs in the facility queue simultaneously. That is, each job receives service for a time which is inversely proportional to the number of competing jobs in the pool. A naive algorithm for simulating the PS discipline is based on a computation and prediction method which doles out an equal amount of service time to all jobs in the pool. This approach suffers in that it is computationally demanding, especially when there are frequent updates of the remaining service time requirements for each job in the pool. To alleviate the computational requirement to some extent, we propose a lazy-update algorithm which accumulates the requisite amount of updating required in a special variable, instead of directly performing the

	Simulated Customers ($\times 100$)				
	10	50	100	200	500
Naive	3.5	17.2	34.1	67.8	169.5
Computational	1.7	8.2	16.3	32.8	83.0

Table 6: Comparison of Execution time (in seconds) for RR algorithms

	Simulated Customers ($\times 100$)				
	10	50	100	200	500
Naive	1.8	8.5	16.5	34.2	86.0
Lazy-Update	1.5	7.4	14.9	29.3	73.2

Table 7: Comparison of Execution time (in seconds) for PS algorithms

update on all jobs exhaustively. When the departure time of the earliest job to leave the pool is to be computed, the value resident in the special variable is subtracted from the remaining service time of the next job to depart, reflecting the true remaining service time. Because of this modification, the only required operations for handling the pool are INSERT and EXTRACTMIN primitives. By combining such infrequent updates with the use of an efficient priority queue data structure, the lazy-update algorithm can be shown to reduce the $O(n^2)$ complexity of the naive algorithm to $O(n \log n)$.

We conduct experiments to measure the execution performance of the proposed RR and PS algorithms, respectively, in comparison to the naive algorithms. We use the single server model for this experiment. In Tables 6 and 7 it can be seen that the proposed algorithms perform well compared to the naive algorithms. The RR discipline exhibits larger performance differences between the naive and computational approaches because of the large number of context-switches required by the former. The differences are not particularly significant for the PS discipline because both approaches exploit computation. A detailed description of these algorithms and related experiments can be found in [18].

5 Conclusions

Our experiences with the the design and implementation of the S_i system have been amply rewarding. The support of a very reliable lightweight process library has greatly reduced the effort required in building an efficient, experimental simulation test-bed. During the

early stages of design and implementation, we faced several different design choices which were at times not altogether consistent with our design principles and objectives. For example, the process scheduler could be implemented by either a dedicated process or by function invocation, with each method leading to different versions of S_i . While the former suffers overheads typically associated with process switching and control, the latter suffers in terms of application-interface inelegance, in that an extra function call is required of a terminating process. Our experiences suggest that, despite the use of lightweight processes, context-switching costs are not insignificant. This is clearly seen in the use of the simple comparison check which eliminates certain unnecessary context switches during simulation execution, improving the performance of both the S_{if} and S_{ip} subsystems. Furthermore, about two-thirds of all overheads are due to thread creation/deletion-related processing within the LWP library, and the rest is due to the layer above the library. Unfortunately, our experimental results show that such overheads can contribute up to 70% of the total simulation time. Thus, simulation would speed up considerably if thread creation and deletion overheads are reduced. To this end, we have embarked on the design of a simple threads library[19] which can provide more efficient thread operations and support a simulator's scheduling disciplines.

With the S_i system we have been successful in developing a software infrastructure which provides an ideal experimental environment. Central to this capability is the modular design philosophy, which unambiguously defines interfaces to the system's functional components. New algorithms can therefore be implemented, tested, and experimented with, almost effortlessly, requiring only the simple need to match interfaces. We have implemented new algorithms in S_i to simulate the round-robin and processor-sharing disciplines, and both the ease of algorithm incorporation in S_i , and the performance of S_i with the new algorithms has been excellent.

A current disadvantage of the S_i system is its lack of portability. This is due to its implementation on a lightweight process library which is machine dependent. However, since S_i adopts a layered design, and because almost all lightweight process libraries support the universal functions of process creation, switching (i.e., yielding), and deletion, the effort required in porting S_i to rest on top of another lightweight process library is minimal. This effort will need to focus only on the process manipulation layer, which is the innermost layer in the S_i system. Therefore, such a modification will be transparent to the application-layer. With the ad-

vent of the standard POSIX threads package (e.g., [14]), we believe that the *S_i* system will be easily portable to any POSIX supporting machine.

References

- [1] G. Birtwistle, O. Dahl, B. Myrhaug, and K. Nygaard. *Simula Begin*. Van Nostrand Reinhold, New York, 1979.
- [2] G. Bruno. Using Ada for discrete event simulation. *Software-Practice and Experience*, 14:685–695, 1984.
- [3] D. Comer. *Operating System Design The XINU Approach*. Prentice-Hall, 1984.
- [4] M. A. Crane and A. J. Lemoine. *An Introduction to the Regenerative Method for Simulation Analysis, Lecture Notes in Control and Information Sciences*. Springer-Verlag, New York, 1977.
- [5] G. Edwards and R. Sankar. Modeling and simulation of networks using CSIM. *Simulation*, 58:2:131–136, February 1992.
- [6] J. B. Evans. *Structures of Discrete Event Simulation*. Ellis Horwood Limited, Market Cross House, England, 1988.
- [7] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, 1978.
- [8] W. R. Franta. *The Process View of Simulation*. North-Holland, Amsterdam, 1977.
- [9] P. Gautron. Porting and extending the C++ task system with the support of lightweight processes. In *Proceedings of USENIX C++ Conference*, pages 135–146, April 1991.
- [10] J. D. Henriksen and R. C. Crain. *GPSS/H User's Manual*, 1983.
- [11] J. Kriz and H. Sandmayr. Extension of Pascal by coroutines and its application to quasi-parallel programming and simulation. *Software-Practice and Experience*, 10:773–789, 1980.
- [12] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools*. The MIT Press, Cambridge, Massachusetts, 1987.
- [13] W. McCormack and R. G. Sargent. Analysis of future event set algorithms for discrete event simulation. *Communications of the ACM*, 24(12):801–812, December 1981.
- [14] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the Winter USENIX Conference*, 1993.
- [15] H. Nakanishi, V. Rego, and V. Sunderam. Superconcurrent simulation of polymer chains on heterogeneous networks. *1992 Gordon Bell Prize Paper, Proceedings of the Fifth High-Performance Computing and Communications Conference: Supercomputing '92*, November 1992.
- [16] V. Rego. A note on two simulation benchmarks. *Simulation Digest*, 20(3):26–35, 1990.
- [17] V. Rego and J. Sang. A remarkable simulation scheduling enhancement. Technical report, Computer Sciences Department, Purdue University, 1992.
- [18] J. Sang, K. Chung, and V. Rego. Computational schemes for efficient simulation of service disciplines. In *Proceedings of 26th Annual Simulation Symposium*, March 1993.
- [19] J. Sang and V. Rego. Xthreads: a simple and efficient thread library. Technical report, Department of Computer Sciences, Purdue University, 1993.
- [20] H. D. Schwetman. CSIM: A C-based process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.
- [21] H. D. Schwetman. Using CSIM to model complex systems. In *Proceedings of the Winter Simulation Conference*, pages 246–253, 1988.
- [22] R. Sharma and L. L. Rose. Modular design for simulation. *Software-Practice and Experience*, 18:945–966, 1988.
- [23] J. Spragins. *Telecommunications Protocols and Design*. Addison Wesley, New York, N.Y., 1991.
- [24] Sun Microsystems, Inc. *SunOS Programming Utilities and Libraries: Lightweight Processes*, March 1990.
- [25] V. S. Sunderam and V. Rego. Eclipse: A system for high performance concurrent simulation. *Software-Practice and Experience*, 21(11):1189–1219, 1991.

The Restore-o-Mounter

The File Motel Revisited

Joe Moran

Bob Lyon

Legato Systems, Incorporated

Abstract

We present a scheme for referencing and accessing saved¹ files in a manner that is transparent to UNIX[®] applications. The scheme requires no kernel modifications. Instead, it uses a "mounted" process that allows users to change directories to the past and browse their saved files with their favorite utilities. The mounted process acts as a protocol gateway between NFS[™] and a commercially available network backup product. Time travel is supported; users may change directories to any moment in the past. Any saved version (not just the most recent version) of any file can be viewed or recovered, even if the file has since been deleted.

Using this transparent method of retrieving saved files by naming their location in the past, a poor man's file migration scheme can be implemented by substituting a symbolic link in to a saved location for a file. Once a file is referenced, the symbolic link can be replaced with its original file. This migration scheme requires no kernel modifications yet remains transparent to UNIX applications and users.

Introduction

This paper describes two file management features² that have eluded UNIX users - file recovery integrated into the operating environment and transparent file archiving (more commonly called file migration). Since saved files can be named and accessed with the same ABI as normal files, users can employ their favorite UNIX tools to find and restore deleted or damaged files. The most obvious examples of such tools are **cd**, **ls**, **find** and **cp**. Equally important though are the users' environments - shells like **csh** or **ksh** that are customized to each individual and provide services such as file name completion and pattern matching, or even windowing environments that completely hide the normal UNIX commands.

File archiving and migration strategies are driven by economics and convenience. Most users do not mind that "inactive" files are removed from their local disks and archived elsewhere provided that the files are easily and rapidly recoverable when needed. Migration of this sort is economical because low capacity, high performance and high priced disks contain only frequently accessed files, while high capacity, low performance and low cost media (like optical disks or tapes) contain seldom or never accessed files. Users find it far more convenient to archive files to free local disk space than to perform the numerous tasks and endure the long elapsed times associated with ordering and installing new and bigger disks.

Customers might also wish to "actively" archive complete directory sub-trees after important events occur at their companies. For example, when a software company ships a major release of its product, it might want to immediately archive that release including documentation, SCCS files, optimized and debuggable objects for numerous platforms and perhaps even the compilers and system libraries used to build the release. A second example is a company that archives a departed employee's home directory, thereby freeing much needed disk space without risking the deletion of important data.

1. We use the word "save" to denote the super set of "backup" and "archive"; save is also easier to conjugate than backup.

2. The features described in this document should not be construed as products features currently available from Legato.

These features are of very limited use if they are not available in the networked environment.

Why delivering solutions is hard

Ease of use is the key attribute to any solution involving file recovery. In the UNIX environment, this usually means the solutions have to be compatible with and transparent to users' existing tools. Since the common ABI for all tools is the UNIX system call interface, this generally means that a solution must be implemented as enhancements to the kernel's filesystem primitives. The Virtual File System [KLEI86], VFS architecture was introduced as part of the implementation of Network Filesystem [SAND85] to separate the filesystem interface from the various filesystem implementations. VFS clearly made adding filesystem functionality easier provided that the implementor had access to the kernel source and understood the rules associated with extending the kernel. However, independent software vendors (ISVs) cannot depend on the VFS interfaces; they are not identical on each UNIX vendor's platform and each new release of a specific UNIX variant can render the ISVs' added functionality useless until the code can be re-ported to the new kernel.

Hardware costs also conspired to keep solutions unavailable. The price of media jukeboxes allowed only the richest and most dedicated companies to consider deploying automated recovery systems. The limited size of this niche market further discouraged UNIX ISVs from producing software that assisted in the automation. However, within the last twelve months, the price of tape jukeboxes has plummeted while the reliability of their robotics has risen sharply. For example, one can buy a 50 gigabyte (before compression) tape jukebox with tape drive for less than \$5,000. With compression and larger jukeboxes, end-user cost might drop as low as three cents per megabyte.

About NetWorker

Legato sells a backup and recovery product called NetWorker [LEGA93]. NetWorker is designed and implemented using a client/server architecture. A server is a machine that manages the saved media (usually tapes or optical disks) and an "on-line index" consisting of two databases. The first database maps a file's key and a time to the file's attributes. The second database maps a file's key and a time to a location in the server's media pool where the complete file's attributes and data are saved. Note that the NetWorker server knows no details about its clients or the clients' files and filesystems. The server is required to retrieve file attributes (quickly) and file data (a little more slowly) when and if a client demands them. The server supports most known tape devices, optical disks, and normal files. Many popular media robots are also supported for "unattended" operation.

The NetWorker client is the machine with files and filesystems that need to be saved. The client walks the local filesystems and sends their descriptions and data to the server. In the case of a UNIX client, the files' keys are both file names and device-inumber pairs, while the files' attributes are identical to NFS attributes. A UNIX directory has no data, but its attributes include a list of all files found within the directory. The client knows nothing about how its files' attributes and data are archived by the server; it only knows that server can deliver either on demand.

The product's client side includes file browsers that allows a user to view saved filesystems as of any time in the past. The command line interface to the browser is similar to BSD `restore -i` with additional features for time specification and file version information. The product also includes X Window SystemTM browsers for command-line challenged users. The browser translates user inputs into queries against the server's on-line index and presents the results as a view into a saved filesystem. Once the user has selected files for recovery, the browser submits a request to the server for data associated with the desired files.

Clients communicate with servers via an application specific protocol built on ONCTM (Sun) Remote Procedure Calls [NOWI88]. The name of the application protocol is "NetWorker Save and Recover" or NSR. Typical protocol stacks are NSR/RPC/TCP/IP or NSR/RPC/SPX/IPX. NetWorker server platforms include eight UNIX vendor's platforms as well as NetWare[®]. Client-only implementations are also available for DOS and numerous other UNIX platforms.

More details of NetWorker's design and implementation are covered in NetWorker's theory of operations manual and beyond the scope of this paper. Most of hard design issues were addressed years ago during NetWorker's devel-

opment, and the features of the restore-o-mounter are built upon these mature solutions.

Ib: the file index browser

When you lose a file, why can't you just "change directory" to the past and copy the file back to the present? This simple question was the inspiration for the restore-o-mounter. The first reason why you can't do that is because the past is not a mounted filesystem and normal users can't use **mount** anyway. However, the automounter [CALL89] mounts filesystems for normal users on demand. The automounter is a UNIX process that mounts itself as an NFS server in the UNIX filesystem. The automounter then turns name references sent from its kernel into mounts to actual filesystems. The automounter also periodically attempts to auto-unmount the filesystems that it mounted.

The index browser (**ib**) takes a similar approach. By default, it mounts itself on the mount point **/ib**, but instead of translating NFS requests from its kernel into mount actions, the requests are translated into browse sessions with NSR servers. Figure 1 shows the conceptual workings of the index browser; it is essentially a protocol gateway that translates from the NFS protocol to the NSR protocol. When presented with a new client name or a new time to browse the saved files, **ib** forks and execs a copy of **iba**, the index browser agent. The generic form of names that **ib** translates to **iba** sessions is

```
[[server:]client][@time]
```

Server is the name of the NSR server to use; note that NSR client software usually determines the appropriate server. The ability to specify the NSR server is needed only in environments where a single NSR client has multiple NSR servers. *Client* is the name of the NSR client whose saved files will be browsed and potentially recovered; the default client name is the local machine's name. *Time* determines which view will be browsed; the default time is the most recently saved version. **ib**'s most visible function is to choose suitable and intuitive defaults and convert human usable time strings into NSR suitable time values. **ib** uses **getdate** routines [BELL87] to provide user friendly time translation. Here are some examples of changing the working directory into some root directories in the past:

```
cd /ib/@now
```

changes directory to the most recent save for the local machine.

```
cd /ib/@yesterday
```

changes directory to yesterday's save for the local machine.

```
cd /ib/ganymede
```

changes directory to the most recent save for the machine ganymede.

```
cd /ib/jupiter:io
```

changes directory to the most recent save for the machine io saved to the NSR server jupiter.

```
cd /ib/jupiter:io@last_month
```

changes directory to the save of a month ago for the machine io saved to the NSR server jupiter.

By the time the **ib** process receives an NFS lookup or stat on a name in **/ib**, the kernel has already concluded that the name was not a filesystem mount point. Therefore, **ib** creates a new directory whose name is derived from the name

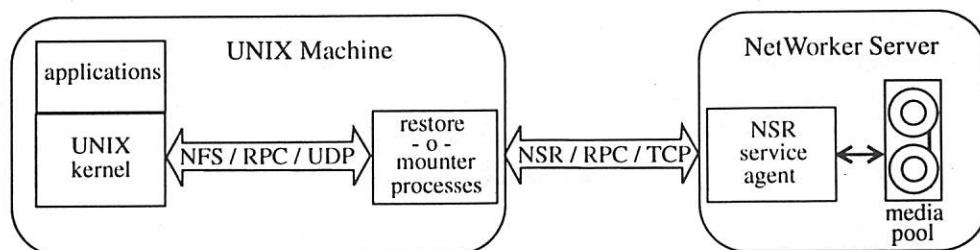


Figure 1. The restore-o-mounter forks processes that translate the NFS protocol originating from the local kernel to the NSR protocol destined for a NetWorker ("backup") server. The NetWorker server is not necessarily a UNIX machine or even an NFS server.

submitted by the kernel, mounts an index browser agent on the derived directory, then creates the submitted name as a symbolic link to the derived directory, and finally returns the symbolic link information to the kernel. For example, after a user references a file from "yesterday's" save of the local machine, **/ib** may look like:

```
io% cd /ib/@yesterday
io% ls -l /ib
total 2
lrwxrwxrwx 1 root 10 Apr 9 10:33 @yesterday -> jupiter:io@04-08-93_10:33:46
drwxr-xr-x 25 root 1536 Apr 6 09:02 jupiter:io@04-08-93_10:33:46
```

One can see that the local machine's name is io and its default NSR server is jupiter. The **getdate** routine mapped "yesterday" to April 8, 1993 at 10:33:46 am, which is exactly twenty four hours prior to the execution of this example.

Ib periodically attempts to unmount the mounts that it automatically performed. Upon a successful unmount, the mount point and symbolic link are removed. We found that removing the mount point caused problems with programs that remember the true mount point path names (e.g., the OpenWindows™ File Manager and **ksh**). To alleviate this, **ib**'s time translator appends or removes an extra underscore character, '_' at the end of the time string when the time string is identical to the string submitted by the kernel. So, the contents of **/ib** may eventually look like:

```
io% ls -l /ib
total 2
lrwxrwxrwx 1 root 28 Apr 9 11:27 jupiter:io@04-08-93_10:33:46 ->
jupiter:io@04-08-93_10:33:46_
drwxr-xr-x 25 root 1536 Apr 6 09:02 jupiter:io@04-08-93_10:33:46_
```

should the unmounted name be re-submitted.

Iba: the index browser agent

Figure 2 shows the results of **ib** forking and exec'ing two index browser agent (**iba**) processes. Like **ib**, **iba** is a "mounted" NFS server process that only responds to calls from its kernel. When launched by **ib**, **iba** mounts itself on top of a directory fabricated by **ib**. **Iba** establishes an NSR connection subject to the arguments passed to it by its par-

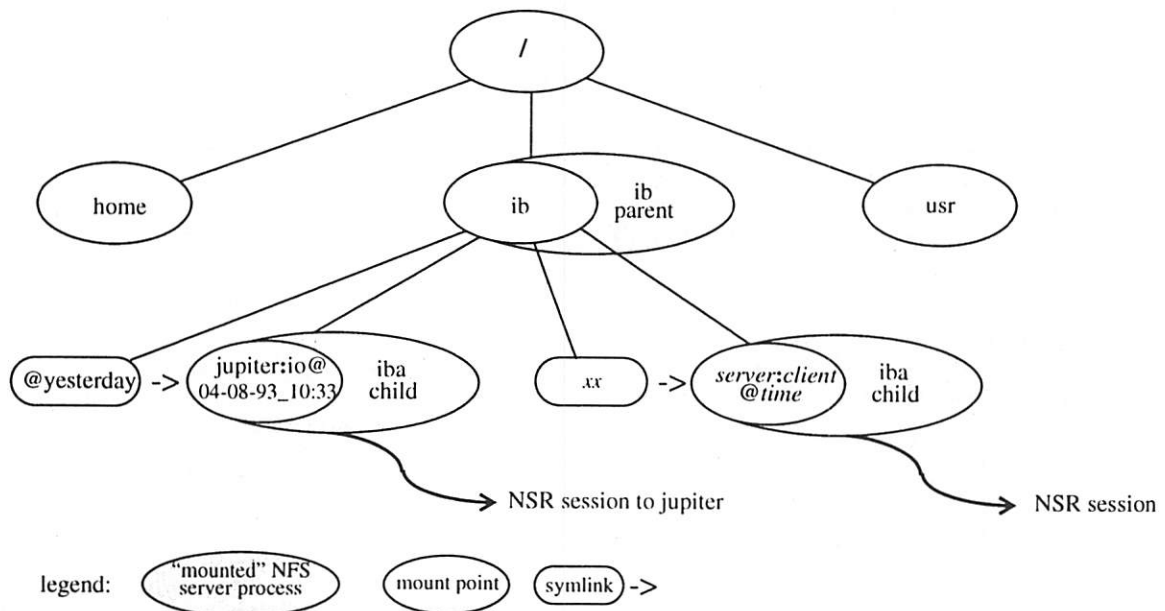


Figure 2. The typical mount points in a UNIX filesystem after two browsing sessions are initiated with the NetWorker server.

ent. The arguments include which NSR server to bind to, which NSR client's index to browse, which view (in time) to present to its kernel, and where to mount itself in the UNIX file name space.

As mentioned before, NetWorker's on-line index for a UNIX file contains the complete stat information of that file. If the file is a symbolic link, then the link's value is also kept on-line; this means that NSR media need not be accessed to resolve symbolic links. Because NFS and NSR file attributes are similar and because the NFS and NSR architectures are similar (since the original designers of both architectures have considerable overlap) **iba**'s primary task is implemented by one-to-one mapping of NFS operations to NSR operations already available in NetWorker's browsers through a library. The challenge in most NFS server implementations is the design of the NFS file handle (fhandle). **Iba** handles contain actual pointers to cached in-core data structures built by the library as files are referenced. **Iba** handles also contain information used to verify their validity. The size and speed of **iba** are covered in a later section.

Iba servicing NFS reads

Iba has command line arguments that determine one of three policies for servicing NFS read requests:

1. Always attempt file recovery on NFS read.
2. Do file recover on NFS read if data can be recovered automatically without requiring any human intervention. This is the default. NetWorker software knows if the volume(s) that contain the requested data are mounted on a device or available in a robotic jukebox. If so, the data is considered "near-line" and **iba** will fetch it. If the data is not near-line, **iba** returns ERREMOTE³ as the results of the NFS read operation.
3. Never attempt file recovery on NFS read. The results of the NFS read operation is always ERREMOTE. This is handy if the customer does not want the NetWorker server bogged down doing actual data recovers and desires a "stat-only" filesystem. In this case, the restore-o-mounter could more appropriately be called the "browse-o-mounter".

The latter options are not found in related work but is useful in the restore-o-mounter, because the NetWorker server most likely stores the data on tape. The seek performance of this class of media is at least three orders of magnitude

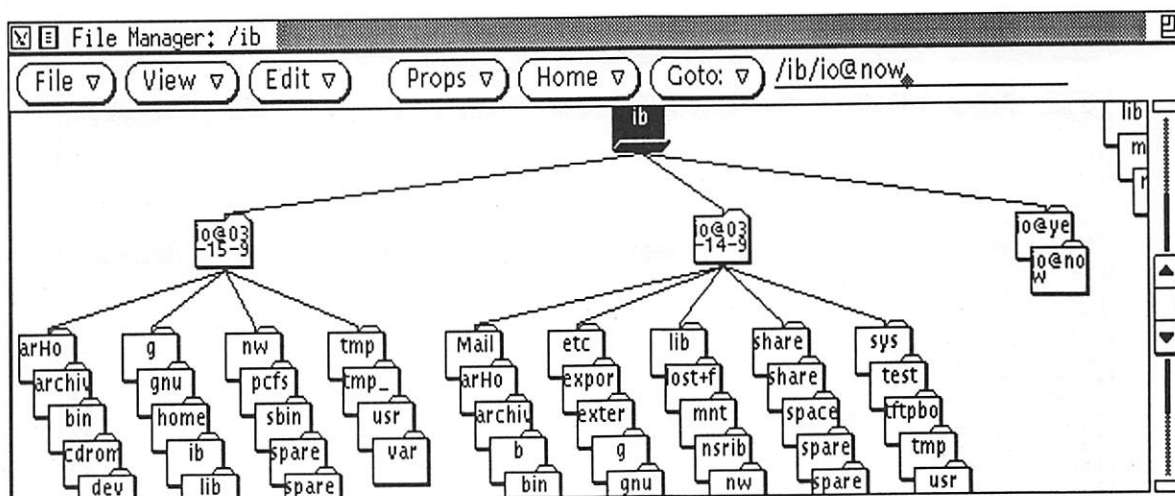


Figure 3. A typical application's view of restore-o-mounted filesystems. The two sub-trees were generated by the user pointing the OpenWindows File Manager to the directories /ib/io@now and /ib/io@yesterday. In this example, numerous directories were deleted between yesterday and today. Note that the two folders at the far right are symbolic links to their corresponding sub-trees.

3. ERREMOTE is not an error defined in the NFS protocol specification. It is defined as part the System V RFSTM definition. Its associated error string "Object is remote" almost matches the desired "the data is too far away". This is probably good enough for a system that was wont to print "Not a typewriter". Clearly **iba** is relying on its NFS client, the local kernel, to pass most errors up to its callers without any attempt to interpret them.

slower than that of rotating media. So, while fetching a few files from tape via **iba** may be practical, **grep**'ing one's entire home directory in the past for a pattern is not recommended. NetWorker's **recover** command is more suitable for moving vast amounts of data from tapes to disks. Still, users agree that a wealth of information is available from stat-only filesystems.

Once a read request is performed, **iba** requests the entire file from the NSR server. The file is then cached in a more traditional filesystem on the UNIX machine. (The location of the file cache is yet another command line argument to **iba**.) The original read operation is not responded to until the entire file is recovered. Subsequent reads are then serviced from the cache. File caching is the optimal way of dealing with the very cheap, but very low performance tape media. However, a subsequent section shows how the cache may become the real thing.

The UNIX kernel is multi-threaded, but **iba** (like most user NFS server processes) is singly-threaded. To avoid suspending all lookup service for the duration of any file fetch, **iba** forks a child to handle the recovery of each file. The parent then resumes NFS service but ignores read requests associated with children who are actively recovering the data. That is, a read request is ultimately answered by the parent **iba**, but only after a child has placed a complete file in the cache.

Versions and hidden files

The NetWorker product provides a means to see every saved version of any file. **Iba** employs hidden file techniques to see and explicitly name these versions. A hidden file name never appears in the results the NFS **readdir** call, but the same name yields successful results when used as arguments to the NFS **lookup** call.

As originally described in The File Motel [HUME88], versions of a file *f* may be found in the hidden directory

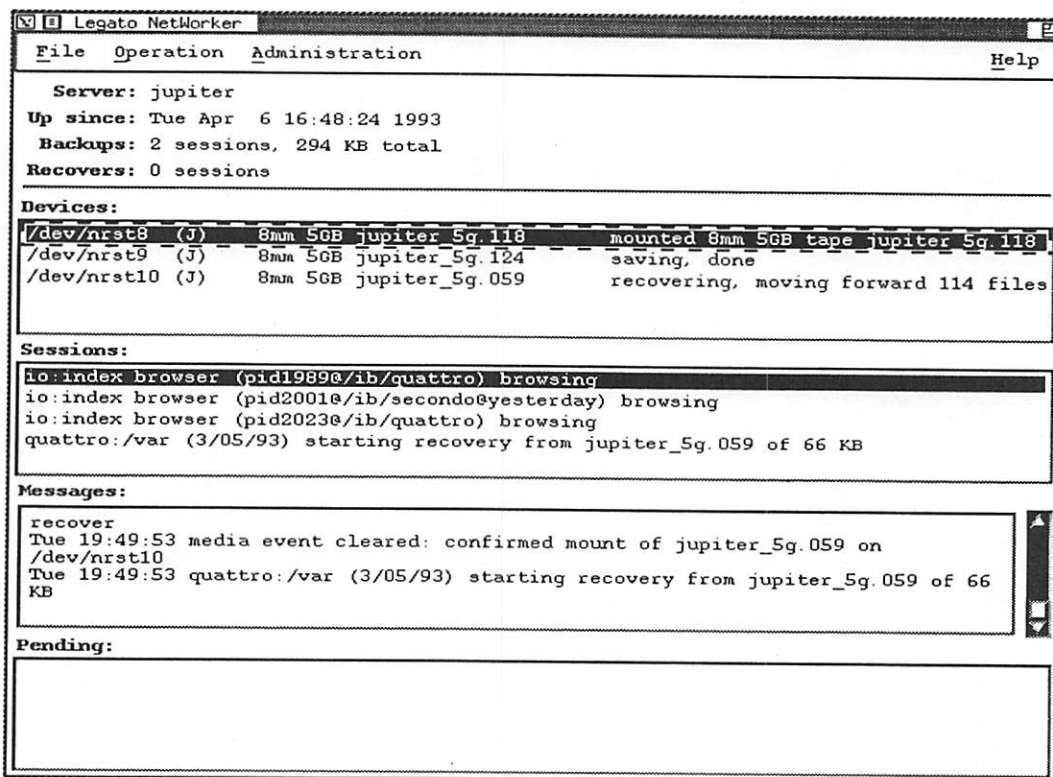


Figure 4. This is a screen shot of a NSR Motif™ based console monitoring the NSR server jupiter. All four sessions originate from two browsing sessions on the machine io. The first browser is navigating the machine quattro's saved files and has forked a child, pid 2023, to recover some of quattro's data (indicated by the last two lines of the Sessions panel). The data may be awhile in coming since 114 tape file marks must first be skipped (indicated by the last line of the Devices panel). The second line of the Sessions panel shows that the browser on io is navigating the machine secondo's filesystems as of yesterday.

f.V, independent of **iba**'s browse time. For example,

```
io% cd /ib/io/etc
io% ls -l rc.local.V
total 28
-rw-r--r-- 1 root 7564 Apr 1 09:03 v1:_jupiter_5g.124_at_\dev\nrst9
-rw-r--r-- 1 root 7396 Mar 5 08:39 v2:_jupiter_5g.113_at_Engn_Jukebox
-rw-r--r-- 1 root 7361 Jan 9 19:58 v3:_jupiter_5g.062_at_Engn_Jukebox
-rw-r--r-- 1 root 7096 Dec 3 12:32 v4:_jupiter_5g.018
```

shows all the saves of the machine io's **/etc/rc.local** file. The generic name of files in a versions directory is

v#:_volume[_at_location]

where **v1** through **vn** are assigned to versions, beginning with the most recent save. *Volume* names the save media containing that version. *Location* is provided if it is known. In this example, the most recent save of **rc.local** is on a tape that is currently mounted in **/dev/nrst9**. In order to have legal file names, **iba** translates any forward slashes in the location information into back slashes. The next two versions reside in the engineering jukebox, while the last version on the tape "jupiter_5g.018" may be sitting in the non-automated portion of the media pool. The listed modifications times above indicate that the file was saved because it changed. Files are also saved during "fulls"; listings of their versions are pretty boring. For example,

```
io% ls -l /ib/ganymede/vmunix.V
total 6401
-rwxr-xr-x 1 root 828881 Jan 11 09:17 v1:_jupiter_5g.124_at_\dev\nrst9
-rwxr-xr-x 1 root 828881 Jan 11 09:17 v2:_jupiter_5g.114_at_Engn_Jukebox
-rwxr-xr-x 1 root 828881 Jan 11 09:17 v3:_jupiter_5g.101_at_Engn_Jukebox
-rwxr-xr-x 1 root 828881 Jan 11 09:17 v4:_jupiter_5g.063_at_Engn_Jukebox
```

shows that the **vmunix** built for ganymede has not changed since its installation. **Iba** usurps all the access time values and replaces them with the (NSR supplied) time when the save occurred. So by using the **-u** option to **ls**, we see

```
io% ls -lu /ib/ganymede/vmunix.V
total 6401
-rwxr-xr-x 1 root 828881 Apr 2 22:25 v1:_jupiter_5g.124_at_\dev\nrst9
-rwxr-xr-x 1 root 828881 Mar 5 22:41 v2:_jupiter_5g.114_at_Engn_Jukebox
-rwxr-xr-x 1 root 828881 Feb 5 21:49 v3:_jupiter_5g.101_at_Engn_Jukebox
-rwxr-xr-x 1 root 828881 Jan 11 21:38 v4:_jupiter_5g.063_at_Engn_Jukebox
```

that ganymede's **vmunix** was saved soon after it was built and late in the evenings of the first Friday of each month thereafter.

Users should approach hidden directories with caution since **pwd** can't find their names:

```
io% cd /ib/ganymede/vmunix.V
io% pwd
pwd: getwd: read error in ..
io% cd ..
```

Iba also supports naming non-directory files at any point in time with the simple syntax *f@time*. These names are also hidden, and they remain independent from **iba**'s browse time. For example, the following two commands are identical:

```
io% strings vmunix.V/v3*
io% strings vmunix@Feb_6
```

Yclept files

The previous section dealt with **iba** hiding versions of files in various directories. A different type of hidden file is one that is hidden by time. This corresponds to the case where users know the name of a lost file, but does not recollect a date (usually in the distant past) when they last had the file. The users could eventually locate their file by sys-

tematically traveling backwards in time until they find the date when the file last existed, but this could be cumbersome.

Iba provides another feature inherited from NetWorker to avoid the tedious searching in time. If a user can name a file, he can recover it. **Iba** requires no special syntax for naming files (including directories) obscured by time. However, the exact name is required since the name does not exist in the current version of its parent directory. Once yclept, files remain legitimate (not hidden) members of their parent directories.

Final words on ib and iba

The file handles provided by **iba** may contain pointers to data structures (nodes) associated with referenced files. These nodes are currently never freed, so the persistence of file handles is guaranteed. Two obvious consequences are that mapping a file handle to its associated data is trivial, but in the worst case may require large amounts of virtual memory. The average measured size of **iba**'s file node is 115 bytes, including malloc overhead. The storage is mostly used by the NFS stat structure; actual file names are a distant second. Because file access through **iba** is casual and because **iba** exits once auto-unmounted, anticipated swap requirements when caching all file nodes is still quite reasonable. Should **iba** memory usage become a problem, an alternative implementation of **iba** can use less caching and place more of a burden on the NSR server by regenerating **iba** file nodes on demand from other information contained in the **iba** fhandle.

The designs of the NFS file handles are different in **ib** and **iba** because these two programs have very different job descriptions, though both are NFS servers. **Iba** is a application level gateway between the NFS and NSR protocols. **ib** provides ease-of-use features by auto-mounting and launching **iba** with the appropriate arguments. The original deployment of NFS separated launch (mount) from operation and it has since proven to be a good idea [PUGS84]. We anticipate other methods for launching the existing **iba** at points spread throughout the traditional UNIX filesystems.

Performance.

The **find** command was used to traverse a machine's filesystems in three ways - local UFS, via NFS from across a quiescent Ethernet™, and via the restore-o-mounter to an NSR server across the same quiescent Ethernet. In all cases, the machines involved are Sun SPARCstation-2™ with 64Mb of main memory and 3.25 inch SCSI disks with 13.5 msec. average access time.

The NSR server managed 416 gigabytes from 34,000 distinct save sessions from 56 clients. The client chosen for the benchmark had 398,000 file instances in NSR server's on-line index. Each of the client's filesystems had at least three full saves (stretching back at least three months) in the on-line index.

Access method	Files found	Seconds elapsed	Normalized UFS	Normalized NFS
UFS, pass 1	71441	621	1.00	0.61
UFS, pass 2	71452	750	1.00	0.60
NFS, pass 1	71284	1120	1.64	1.00
NFS, pass 2	71286	1144	1.67	1.00
IB, pass 1	70979	1890	2.76	1.67
IB, pass 2	70979	479	0.70	0.42

Figure 5. The relative times needed to run the **find** command against filesystems accessed via UFS, NFS, and the restore-o-mounter's index browser processes.

The identical **find** command is run twice to show the caching effects of UFS, NFS and the restore-o-mounter. Figure 5 shows the results. Note that NFS takes 64% - 67% more time than local UFS. In the first pass, the restore-o-mounter takes 67% more time than NFS, or 176% more time than UFS. Most of the restore-o-mounter time for the first pass is attributable to the NSR server, and not to the **iba** process. The second time the command is run yields little difference in the performance of UFS and NFS. But the restore-o-mounter blazes away, taking only 42% of the time that NFS takes and only 70%⁴ of the time that UFS takes!

Each pass of the **find** benchmark shows the extremes of the relative lookup and stat performance of the restore-o-mounter. Neither are very realistic given the usage model of the restore-o-mounter. Read performance was not benchmarked because it is highly dependent on the speed of the underlying media holding the file data. Obviously, minutes may pass before any data is forthcoming from a tape jukebox. Related work asserts that these long lag times are reason enough to exclude tape media from any serious consideration of transparent file recovery services. Our experience does not bear this out; users tolerate seemingly long delays because the process is entirely automated, highly reliable, and provides feedback (via the NetWorker monitors) regarding its progress. Once users trust the system, they usually switch to another task (take a coffee break or read e-mail) and switch back after their files are recovered.

Transparent file migration

We have shown how the restore-o-mounter provides application transparent file access to saved files. In this section we present a new application that exploits restore-o-mounter features to provide a poor man's file migration utility.

Note that NSR clients distinguish backup data from archive data. The NSR server separates these two types of data into different media pools. The media with backup data is eventually deleted or recycled according to customer policies; after all, the data is merely a copy of the real thing.

Archived data is expected to live forever. Rather than being a copy, it is assumed that the data is the real thing. As the NetWorker product ships today, files are archived from explicit user actions. Once the files are archived, users may then choose to explicitly remove⁵ some files. In doing so, it is up to the users to remember their actions and to recover the files should they ever be needed.

The best way to remember removed files is by systematically making notes about them in the filesystem. Related work incorporates new information into a file's inode. This has the advantage that users or applications need not perform any special actions to recover removed files when running a modified kernel that automatically reacts to references to the new type of inode.

Our scheme also makes notes about removed files. But rather than extending the inode information, a removed file is simply replaced by a symbolic link to the restore-o-mounter name of the archived file. For example, a user may record all his outgoing e-mail messages in a file that he periodically moves to his notion of a mail archive. If the full path name of such a file was **/home/io/mojo/mail.record.92**, then the associated symbolic link might appear as:

```
mail.record.92 -> /ib/jupiter:io/home/io/mojo/mail.record.92@03-03-93_11:52:47
```

Note that the resolution of the symbolic link contains the explicit save time of the archived file. Omitting the time would cause an infinite loop once the symbolic link itself is saved and the symbolic link within the **iba** filesystem is dereferenced! The following symbolic link is functionally equivalent:

```
mail.record.92 -> /ib/jupiter:io@03-03-93_11:52:47/home/io/mojo/mail.record.92
```

But, while the first link can use any existing **iba** session for the client **io** to the NSR server **jupiter**, the second requires an explicit **iba** session at the time 03-03-93_11:52:47. Therefore, the first style of link is used to avoid unnecessary process forking.

4. The biggest advantage that the restore-o-mounter has over the traditional filesystems is that its files are static and its metadata can be cached in a small amount of virtual memory. Never-the-less, we hope this astounding number helps to debunk three myths: code runs faster in the kernel; marshalling data through XDR is inefficient; context switching to user level (file) services is expensive.

5. In some of our competitors products, file removal is automatic. This is euphemistically called "filesystem grooming".

Two nice features fall out from the symbolic link approach to migrated files. The first is that user may rename their files without losing the associated archives. Second, the symbolic links are saved during traditional backups. So if an entire directory or entire disk is lost, the traditional recovery replaces the symbolic links; all the archived data is not placed back onto the disk.

Unmigration

As mentioned before, **iba** usually caches a complete file's data when it processes an NFS read request. The restore-o-mounter can be invoked with an option which recovers files back to their original locations when the files' data is accessed. The following conditions must be met if **iba** is to recover a file in place:

- the file name presented to **iba** must be of the form *f@date* where *date* is exact match of *f*'s save time
- the file cannot overflow the filesystem's space
- the original location of the file must still be a symbolic link that directly resolves to the file name presented to **iba**

The last rule implies that renamed migrated files cannot be recovered in place. Heuristics could be added to **iba** to allow it to hunt down the renamed symbolic link and recover to it. A simple heuristic would be to look for the symbolic link in the original parent directory. A more complicated one could be to have the NetWorker **save** command build a special symbolic link cache as it performs its daily backups. (If the file is recovered from tape, then **iba** may have quite a bit of clock time to burn; searching for the renamed symbolic link could be accomplished then.)

Use of the **-L** option to **ls** (use **stat** instead of **lstat**) hides the fact that a file is migrated:

```
io% cd /home/io/mojo
io% ls -lLt mail.record*
-rw----- 1 mojo 985484 Apr 10 23:42 mail.record
-rw----- 1 mojo 4871761 Jan 3 1993 mail.record.92
-rw----- 1 mojo 6720668 Jan 2 1992 mail.record.91
```

while no **-L** option to **ls** indicates the files' real status:

```
io% ls -lt mail.record*
-rw----- 1 mojo 985484 Apr 10 23:42 mail.record
lrwxrwxrwx 1 mojo 57 Mar 3 11:54 mail.record.92 -> /ib/jupiter:io/home/io/
mojo/mail.record.92@03-03-93_11:52:47
lrwxrwxrwx 1 mojo 57 Mar 3 11:54 mail.record.91 -> /ib/jupiter:io/home/io/
mojo/mail.record.91@03-03-93_11:52:47
```

Here we use **egrep** to recover in place a file from a tape jukebox, and use **ls** to see its new status:

```
io% /bin/time egrep presto mail.record.92 > /tmp/foo
253.8 real 5.7 user 2.1 sys
io% ls -lt mail.record*
-rw----- 1 mojo 985484 Apr 10 23:42 mail.record
-rw----- 1 mojo 4871761 Jan 3 1993 mail.record.92
lrwxrwxrwx 1 mojo 57 Mar 3 11:54 mail.record.91 -> /ib/jupiter:io/home/io/
mojo/mail.record.91@03-03-93_11:52:47
```

The filesystem that **iba** exports for recover in place is mounted for read and write access since a file could be modified by applications. The applications may continue to access the file via its NFS handle into the **iba** process while subsequent opens of the file will be handled by the file's originating filesystem. Thus one can modify a currently migrated file and have the correct behavior (e.g., **cat >> currently_migrated_file**).

The final observation is that in place recovery can be performed by a restore-o-mounter running on any machine. The example above was performed on the machine **io** which was also the originator of the migrated files. The example could have been performed on **ganymede** which NFS mounts **io:/home/io**. This demonstrates why both the NSR server and NSR client are named in every symbolic link.

Migration

Readers may have resigned themselves to the fact that this paper only addresses recovers and not saves. Despair no more. We allocate the following half page to describing archive saves.

The **migrator** was implemented by modifying the NetWorker **save** command to declare its saves as archive data. Then normal save policies were replaced with migration policies. Policies include:

- the file's type
- the file's change time
- the file's size
- the file's owner and permissions
- the number of exact copies of the file on archive media

Only regular files are automatically migrated. Files greater than 16 Kbytes were typically chosen. Some systems also provide upper bounds for candidate files. Filtering files owned by special users or with certain permissions is also a good idea. For example if we migrate an executable owned by root, then we may end up migrating the kernel, shared libraries, single-user utilities like **fsck**, and even **iba** itself.

Some users feel that the file should be archived on multiple media before it is replaced with a symbolic link. The **migrator** checks this by reading the versions directory of a candidate file as it is presented by **iba**. If there are enough duplicate versions located on different volumes, then the **migrator** replaces the file with a symbolic link to the last such version. If a symbolic link is not substituted for the file but the file is a candidate for later migration, the **migrator** then saves the file to archive media. To actually migrate a file, the **migrator** must visit the file at least twice. The first time it saves the file to archive media; the last time, it replaces the file with the corresponding symbolic link.

Caveats

The **migrator** is an application that we expect to traverse the filesystems once a day, once a week, or maybe only once a month. A daemon that polls for the free space in filesystems could be implemented to launch a **migrator** should the free space drop below some threshold. Without kernel support, the **migrator** cannot protect applications from ENOSPC errors caused by short term anomalous behavior.

Recovery in place of archived files by NFS clients can only succeed if the NFS clients shares a common view of mounted filesystems with the NFS server.

Re-migrating a recovered in place but unmodified file may occur; this is not desirable. The unnecessary re-migration could be avoided if **iba** could control a restored file's change time.

Related work

Several earlier systems have provided a filesystem interface with time travel capability (primarily to access versions from past backups).

The restore-o-mounter was originally inspired by The File Motel [HUME88] which implemented a Version 8 File System in addition to its own access commands to browse and recover past backups to optical disks. The restore-o-mounter provides additional functionality by extending the name space to allow specification of the NSR server, NSR client, and browse time. It also allows specification of an arbitrary version of any file within the restore-o-mounter filesystem and includes in place recovery of archived files.

The 3DFSTM [ROOM92] uses NFS access to a filesystem constructed from previous backups to an optical disk jukebox. 3DFS has many similarities to the restore-o-mounter and NetWorker, but has a number of differences in the design and implementation.

- Both 3DFS and the restore-o-mounter look like a filesystem using a standard NFS interface. Both 3DFS and the

restore-o-mounter allow you to use unmodified UNIX commands to browse and access old versions of files. Both accept dates for files at any point in the filesystem hierarchy. While the restore-o-mounter restricts the dates for a directory tree to the top of the tree (i.e., a mount point), 3DFS allows time travel at arbitrary points in the tree. But, the greater flexibility 3DFS may present a less consistent view of the filesystem name space.

- 3DFS uses only optical disk jukeboxes. The restore-o-mounter (via NetWorker) can use a variety of save media (e.g., 8mm and 4mm tapes, optical disk) with or without jukebox support.
- 3DFS uses remote NFS mounts to a dedicated server, while the restore-o-mounter uses only local NFS mounts to a gateway process. Using local NFS mounts makes it easy to figure out when a mount is no longer busy (the unmount system call doesn't return EBUSY), provides control of NFS access (e.g., **ib** sets up reasonable timeouts and retransmission parameters), and makes it possible to provide better file migration support (it can recover files in place). The disadvantage of using local NFS gateway processes is that there can be no centralized network-wide caching of recovered files (although this functionality can be provided by the NetWorker server).
- 3DFS has more complicated naming rules and **glob**'ing rules for increased flexibility, but sometimes requires special commands. Each directory can have a date associated with it, but this date is not visible via the standard UNIX **pwd** command. The restore-o-mounter has simpler file naming conventions and doesn't require any special commands to access version information, but provides less flexibility for accessing the data. We view the less complicated naming conventions and fewer options as a benefit.
- 3DFS internal access methods are strictly path name based, so it does not handle file or directory rename situations gracefully. For example, renaming a directory renames all the files under it and breaks their history. The restore-o-mounter (via NetWorker) uses both path name and file id access methods and properly handles rename situations. This allows a correct view of a filesystem as of any time (subject to the times of when saves are performed).

The Plan 9 filesystem [PIKE90] uses a true filesystem on optical disk and a two-level cache to provide transparent filesystem access to previously backed up files. It also provides automatic file migration. However, Plan 9 provides support only for files in the Plan 9 filesystem while the restore-o-mounter works with any traditional UNIX filesystems.

The Inversion filesystem [OLSO93] uses the POSTGRES database system to provide fine grained time travel and transactional operations. However, to take advantage of these facilities applications currently need to be rewritten to use new programmatic interfaces provided by a special library that must be linked with each application.

A number of systems provide a more tightly integrated file migration solution by providing custom OS's or modifying the kernel. The BUMP project used kernel modifications to provide automatic file migration hooks [MUUS89]. Epoch and NetStor provide file migration products based on NFS servers. Epoch has traditionally used specialized file servers to provide automatic file migration services. Currently Epoch is moving away from specialized kernels and is working to define UNIX kernel hooks to allow for user processes to handle the bulk of the file migration services [WEBB93]. The restore-o-mounter file migration philosophy differs from these approaches in that it requires absolutely no OS changes at the cost of some functionality.

Conclusions

We have shown that both transparent file recovery and transparent file migration can be accomplished with no modifications to the kernel. We avoided the VFS abstraction because an ISV cannot take advantage of it when delivering products into the ever changing UNIX markets. A far more stable and universal interface, the NFS protocol was readily combined with our NSR protocol to provide the new functionality. Symbolic links, often reviled by users, were exploited as stubs for migrated files, making our scheme transparent to any application and compatible with any backup method.

The Automounter philosophy is useful for moving functionality out of the kernel and providing new privileges for normal users. We believe that extending the UNIX system via new processes will soon become the norm and not the exception.

Finally, the architecture described in this paper allows a variety of devices (not just optical jukeboxes) to be practical for both backup and file migration.

Acknowledgments

Joe Moran implemented all aspects of the restore-o-mounter after Bob Lyon said it should be easy. Bob Lyon wrote most of this paper. The restore-o-mounter builds upon the NetWorker product which is implemented and deployed by the team at Legato Systems.

Dave Cohrs, Bill Nowicki and Linda Weinert helped to turn our random thoughts into mostly coherent English.

We give special acknowledgment to Tom Lyon for envisioning and championing the mechanisms that allow NFS servers to do more than just share files.

References

- [BELL87] Steven M. Bellovin is the author of **getdate** routines, public domain software written in 1987 acquired from the University of North Carolina at Chapel Hill.
- [CALL89] Brent Callaghan and Tom Lyon, *The Automounter*, 1989 Winter (San Diego) Usenix Conference Proceedings.
- [HUME88] Andrew Hume, *The File Motel - An Incremental Backup System for Unix*, 1988 Summer (San Francisco) Usenix Conference Proceedings.
- [KLEI86] Steven R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, 1986 Summer (Atlanta) Usenix Conference Proceedings.
- [LEGA93] Legato Systems, *NetWorker Product Overview*, 1993, Palo Alto, CA.
- [MUUS89] Michael John Muuss, Terry Slattery, and Donald F. Merritt, *BUMP - The BRL/USNA Migration Project*, November 1989 LISA Conference Proceedings, Monterey, CA.
- [NOWI88] William I. Nowicki, *RPC: Remote Procedure Call Protocol Specification*, RFC 1057, Network Information Center, USC ISI, Marina del Rey, CA, 1988.
- [OLSO93] Michael A. Olson, *The Design and Implementation of the Inversion File System*, 1993 Winter (San Diego) Usenix Conference Proceedings.
- [PIKE90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, *Plan 9 From Bell Labs*, 1990 Summer UKUUG Conference Proceedings, London.
- [PUGS84] Tom Lyon, *Why separate mounts from the NFS service*, 1984, private communication to Sun's Network File System's group.
- [ROOM92] William D. Roome, *The 3DFS: A Time-Oriented File Server*, 1992 Winter (San Francisco) Usenix Conference Proceedings.
- [SAND85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, *Design and Implementation of the Sun Network Filesystem*, 1985 Summer (Portland) Usenix Conference Proceedings.
- [WEBB93] Neil Webber, *Operating System Support for Portable Filesystem Extensions*, 1993 Winter (San Diego) Usenix Conference Proceedings.

Trademarks

The authors have made every effort to supply trademark information about products and services mentioned in

this paper. 3DFS, Ethernet, Motif, NetWare, NFS, ONC, OpenWindows, RFS, SPARCstation-2, Sun-3, SunOS, UNIX, and X Window System are trademarks of their respective companies.

Biographies

Joseph Moran is a principal engineer at Legato Systems Incorporated which he cofounded in 1988. At Legato, he has had major roles in architecting and implementing all of Legato products including Prestoserve™, the client and server sides for UNIX and NetWare versions of NetWorker, and NetWorker for DOS in addition to pitching in where ever needed. From 1984 to 1988 he was a Member of Technical Staff at Sun Microsystems. While at Sun he worked on a variety of tasks for the SunOS™ kernel including implementing the virtual memory system, doing the original Sun-3™ SunOS port, bringing SunOS up on a number of new machines, and designing and implementing **kadb** after getting tired of debugging kernels on bare hardware. From 1982 to 1984 he was a System Programmer at Hewlett Packard where he was involved with various UNIX related activities including working on the original BSD UNIX port to the Hewlett Packard Precision Architecture. He received a M.S. in Computer Science in 1982 and a B.S. in Electrical and Computer Engineering in 1980, both from the University of Wisconsin, Madison. He can be reached via e-mail at mojo@Legato.COM.

Robert B. Lyon is the Vice President of Core Technologies at Legato Systems Incorporated which he cofounded in 1988. At Legato, he has exerted architectural influence on the implementation and deployment of the company's products, and at times, implemented components like NetWorker's database for its on-line file indices. From 1983 to 1988 he was the Project Leader and Manager of Sun Microsystems Network File System group where he played a similar role after implementing Sun's Remote Procedure Call and eXternal Data Representation (RPC/XDR) package. From 1979 to 1983 he was a Member of Technical Staff at Xerox Corp Systems Development Division where he assisted in the implementation and tuning of all levels of the XNS protocol stack and had project lead responsibilities for the Clearinghouse Name Service. Prior to Xerox, he was an MTS at Bell Labs in Holmdel NJ where he had system administration responsibilities for a small lab of UNIX machines; he claims to be the first to deploy **uucp** outside of Murray Hill. He received a M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Engineering from Cornell University in 1977. He can be reached via e-mail at blyon@Legato.COM.

The Autofs Automounter

Brent Callaghan, Satinder Singh

SunSoft Inc.

Abstract

Prior to the introduction of the automounter in 1987, NFS mounts were administered separately on each workstation. The automounter has provided administrators with a tool to construct a filesystem namespace that can be shared across an organization. While the automounter is widely used, its success has been tempered by problems inherent in its implementation. This paper describes a new implementation of the automounter based on a new filesystem. This new automounter not only fixes the problems, but provides some interesting opportunities for future development.

1.0 Introduction

An automounter is a service that automatically mounts filesystems upon reference. This service is particularly useful for accessing NFS filesystems. An automounter automatically mounts filesystems without requiring the workstation user to become superuser and use the mount command. To a user, all filesystems appear to be immediately, and continuously available.

There are two automounters in wide use. The first to become available was the automounter in SunOS 4.0. This automounter is available to ONC licensees and is available in many Unix system platforms including those from Sun, IBM, DEC, HP and SGI. The Amd automounter is included in BSD 4.4 and has been ported to almost every Unix system. While these automounters have different command and map syntax, they share a fundamental property: they are implemented as NFS servers [2, 5, 6]. The server is a daemon process on the client that is NFS mounted at directories where its services are required. At each mountpoint the kernel communicates with daemon just as it would with a remote NFS server to the extent that it will print "server not responding" messages if the daemon is busy or dies. The daemon receives NFS messages from the kernel for any filesystem activity directed at these mountpoints. This allows the daemon to interpose its services [1]. Each mountpoint is associated with a map that determines what names appear under the directory and describes the filesystems they correspond to. On receiving a request for a filesystem that is not yet mounted, the daemon uses the map information to mount the filesystems in another directory (/tmp_mnt or /a) and returns a symbolic link to this mountpoint in response to the NFS request. While the filesystem is mounted continues to return the symbolic link in response to NFS requests. After a period of inactivity, the daemon can choose to unmount the filesystem. It is beyond the scope of this paper to describe all automounter features or the syntax of its maps. Please refer to references [1 - 6] at the end of the paper for this information.

1.1. Problems

While implementing an automounter as an NFS service is an attractive way to provide a valuable service without requiring any kernel changes, it, this presents some serious problems:

Symbolic links

A single automount daemon can service many mountpoints. At each mountpoint the daemon associates a map, and emulates either a symbolic link (direct map) or a directory of symbolic links (indirect map). The symbolic links point to a directory where the automounter performs NFS mounts - /tmp_mnt or /a.

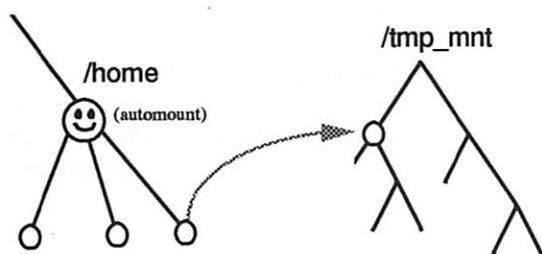


Figure 1. The smiley face represents a mountpoint of the automounter's daemon. Beneath this mountpoint the automounter maintains a directory of symbolic links that point to NFS mounts beneath the /tmp_mnt directory. On receiving a lookup for a link which does not exist, the automounter consults the indirect map associated with the directory and uses the information from the map entry to do an NFS

The automounter unmounts what it considers to be "idle" mounts – those that have not been active – or that can be unmounted (not busy). As long as all references to these mounts are made through the daemon mountpoint, the daemon can replace the mounts as necessary.

However, if a process invokes the `getwd()` function to obtain the path of the current directory while in an automounted filesystem, it will obtain a `"/tmp_mnt/..."` path. If this "back door" path is cached and used sometime later, there is no guarantee that the filesystem will still be mounted there. The automounter cannot detect references to empty mountpoints unless they are made through the automounter's mountpoint. A common victim of this behavior is the `at` command. It uses the `pwd` command to record the current directory so that it can be `cd`'ed to for subsequent invocation of the script. The symbolic links also confuse users because `"/tmp_mnt"` frequently appears as a prefix to the current directory.

Various workarounds have been proposed for this problem. The most common was for the `getwd()` function to strip prepended `"/tmp_mnt/"` from paths. This workaround didn't take into account the effect of the automounter's `-M` flag that allowed users to specify a directory other than `"/tmp_mnt/"`. It is also questionable whether the semantics of `getwd()` should be changed this way.

The symbolic links present a problem for relative references between separate automounter mounts. For instance given the two directories `/home/bob` and `/home/carol` it seems reasonable that Bob should be able to `"cd ../carol"` but this will fail if Carol's directory isn't already mounted.

Adding Automounter Mountpoints

The set of automounted mountpoints can be specified only when the automounter is started. If a new mountpoint is added to the master map, the automount daemon must be killed and restarted. The automounter can be terminated with a `SIGTERM` signal. It catches this signal, attempts to unmount itself from its mountpoints and also to unmount mounts in `/tmp_mnt`. If all these unmount attempts succeed, then restart is possible. However, this interruption in automounter service is very disruptive.

Performance

A pathname that passes through an automounter mountpoint prompts the kernel to contact the automounter and retrieve the symbolic link that points into `"/tmp_mnt/"`. While these symbolic links are returned more quickly than they would be from a remote NFS server, there is still considerable overhead in generating remote procedure calls and context switching between the kernel and automounter.

Multiple Threads

The automounter daemon is single-threaded. While performing a mount for one process, other processes must wait until the mounting task is complete. As an RPC client, the automounter is subject to network timeouts that may deny service to *all* processes for inconvenient periods of time that users perceive as a "hang". A single-threaded automounter is also prone to deadlock. If in servicing a mount request the daemon or a child process stumbles into another automounter mountpoint then deadlock is certain. Much code in the current automounter is devoted to detecting and avoiding simple deadlock situations.

The Amd automounter provides higher availability by forking mount attempts and by using its own asynchronous RPC's. Since `fork` is a heavyweight operation, it exacts a price in performance. Asynchronous RPC's provide a way to avoid blocking on explicit RPC calls, but synchronous RPC calls embedded in system libraries are still a risk.

1.2. In-place Mounting

Several of these problems could be fixed by having the daemon perform mounts at the point of reference instead of using a symbolic link to redirect references to another mountpoint. Without the symbolic links, there would be no "back door" path and performance would improve since the daemon would not need to service requests for the symbolic links while the filesystem is mounted. We examined this alternative implementation, but could not resolve a serious deadlock problem. Any filesystem operation on a directory that requires the daemon to perform a mount must be blocked until the mount is complete – but in order to perform a mount the daemon risks blocking at the same directory. We could not think of a way around this problem within the context of the daemon as an NFS server.

The remaining sections describe our re-implementation of the ONC automounter based on a new kernel virtual filesystem.

2.0 The Autofs Filesystem

The *autofs* is a virtual file system [8] (VFS) that supports automounting with the support of an external daemon. Since it is required to support only a small subset of VFS operations, it is a minimal VFS.

The autofs filesystem intercepts requests to access directories that are not present and calls an external daemon *automountd* to mount the requested directory. The automountd locates the filesystem, mounts it within the autofs and replies. On receiving the reply, the autofs allows the blocked request to proceed. Subsequent references to the mount are redirected by the autofs – no further participation is required by the automountd.

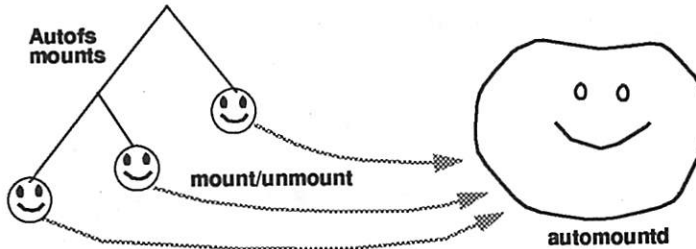


Figure 2. A single automount daemon provides a filesystem mount and unmount service to a number of autofs filesystems. Within an autofs directory, references to filesystems that need to be mounted are conveyed as remote procedure calls to the automount daemon which mounts requested filesystems on or beneath the autofs directory. After a period of inactivity the autofs requests the daemon unmount the filesystems it

2.1. An Autofs Mount

The automount command adopts a new role of mounting and unmounting autofs mounts. The automountd daemon is completely independent. The automount command reads the master map *auto_master* that contains a list of the autofs mounts. It compares this list with the list of autofs mounts from the list of mounted filesystems in */etc/mnttab* and adds or deletes autofs mounts as necessary. It will also remount existing autofs mounts if mount information has changed e.g. the name of the map or the default mount options.

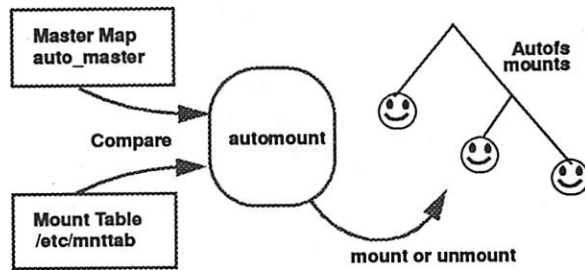


Figure 3. The automount command is used to keep autofs mounts synchronized with the master map, adding, deleting or modifying autofs mounts as necessary. At system start-up the command is run to create the initial set of autofs mounts. The command can be run anytime thereafter to reflect modifications

An autofs mount requires the following information:

1. The name of the map to be associated with the mountpoint, e.g. the autofs mount on */home* uses a map called "auto_home".
2. The address of the user-level daemon for remote procedure calls.
3. An indication as to whether the automatic mounting is to be direct – or indirect.
4. Default options to be used for mounting e.g. *ro,nosuid*

An autofs mount appears in the mount table with a filesystem type of "autofs" e.g.

```
auto_home /home autofs ignore,indirect
```

2.2. Automatic Mounting

The autofs maintains an illusion of continuously available filesystems. When it receives a request to access a non-present filesystem through a *getattr* or *lookup* operation, it calls the *automountd* which mounts the requested filesystem. Upon receiving a successful reply the autofs directs the *getattr* or *lookup* to the new mount. While the mount is in place, further accesses are directed to the new mount without contacting the daemon.

The autofs supports the notion of *direct* and *indirect* mounts implemented by the old automounter.

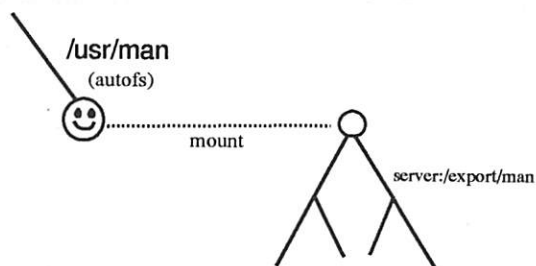
2.3. Direct Mounts

This is the simplest and most intuitive form of automatic mount. With zero *offset* (see section 2.5.) daemon mounts overlay the autofs mountpoint. On receiving a request to access its mountpoint, the autofs calls the automountd giving its path and a direct map name, e.g.

```
/usr/man    server:/export/man
```

The daemon mounts over the autofs mountpoint and replies. On receiving the reply, the autofs redirects the blocked request to the new mount. When the mount is unmounted, the autofs mount is exposed again.

Figure 4. Direct mounts. A direct mount with no offset (see section 2.5.) overlays the autofs mountpoint. This appears in the system mount table (/etc/mnttab) as two mounts at the same directory. These mounts must be performed with an "overlay" flag to override a SVID requirement that mounts cannot be stacked.



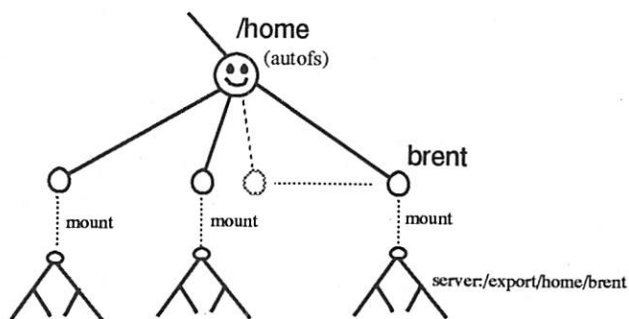
2.4. Indirect Mounts

At an indirect mountpoint the autofs provides a directory of automatic mounts. On receiving a request to lookup a name in this directory, the autofs calls the automountd with the subdirectory name and map name, e.g.

```
brent      server:/export/home/brent
```

The daemon looks up the name in the map, mounts the corresponding filesystem and replies to the autofs which redirects the blocked request to the new mount. While the filesystem is mounted accesses are directed by autofs to the mountpoint - communication with the daemon is not required.

Figure 5. Indirect mounts. The autofs filesystem is required to support a directory of mountpoints. A *readdir* request will show just filesystems that are already mounted. A lookup request for a name that is not in the directory but is in the map will cause the name to be added to the directory when the mount is complete.



A *readdir* request on the autofs directory will return only a list of existing mounts. The names of potential mounts cannot not be displayed. This restriction prevents unintentional mounts e.g. "ls -l /home/*" would otherwise cause every home directory to be mounted. This requirement continues to be an obstacle for file browsers.

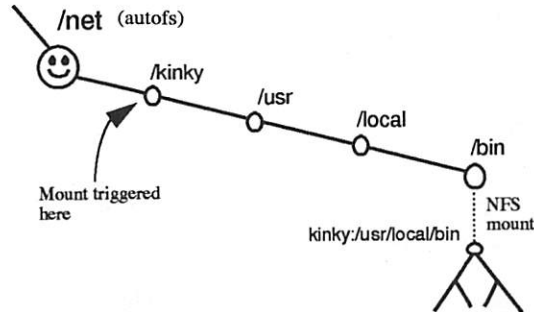
2.5. Offset Mounts

The automountd may need to mount several directory levels below the direct or indirect mountpoint. In this case the mount is offset by some number of directory levels from the directory that triggered the automatic mount. The autofs is required to provide as many directory levels as necessary to account for the offset. For example: assume the following automount map entry:

```
kinky      /usr/local/bin    kinky:/usr/local/bin
```


If used with an indirect mountpoint of `/net` then `/net/kinky/usr/local/bin` would refer to `kinky:/usr/local/bin`. The mount is on `/net/kinky/usr/local/bin`. It requires that the autofs provide the directories along the path `kinky/usr/local/bin/`.

Figure 6. An offset mount requires one or more directories be created within the autofs filesystem to create a path to the mountpoint.

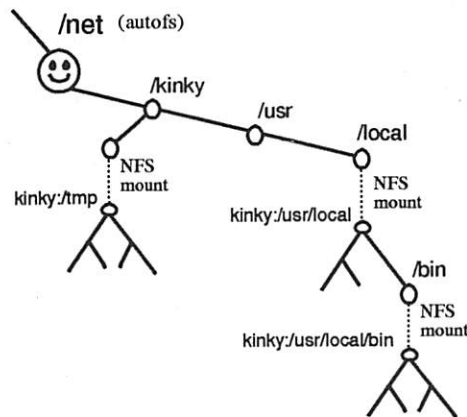


Offset mounts may be required for direct or indirect mounts. The automountd is responsible for constructing the required directories within the autofs filesystem. The autofs supports the `mkdir`, `rmdir`, `lookup`, and `getattr` operations to allow creation of these offset paths.

2.6. Multiple Mounts

To satisfy a mount request, the daemon may need to mount more than one filesystem. A common example of this is the automounter's `/net` facility. It attempts to mount all the exported filesystems from a server when `/net/servername` is referenced. The daemon may be quite busy building offset paths and doing mounts before it responds to the autofs request. Note that some of these mounts may be hierarchically related – one within the other.

Figure 7. Several filesystems may need to be mounted as a hierarchy. The automount daemon creates the offset paths as necessary to construct mountpoints within the autofs filesystem. The daemon responds with a successful mount complete message if at least one of the filesystems has been mounted. This facility is most commonly used with `/net` mounts that attempt to reproduce an NFS server's exported hierarchy on the client.



2.7. Deadlock Issues

The autofs is required to block access to absent filesystems until the required mounts are completed by the automountd. The automountd signals completion with its reply. In order to satisfy an autofs request, the automountd may be required to make directories and perform mounts within the autofs without itself being blocked. The autofs must not block requests from the daemon. How does the autofs identify requests that should not be blocked?

The automountd modifies pathnames by appending a space. This space acts as a flag to the autofs indicating that the request should not be blocked. This solution satisfies the requirements in the previous paragraph. The trailing space on the name of a mountpoint is invisible in `/etc/mnttab` entries since it is parsed as whitespace. The following scenario demonstrates this process:

1. A process attempts to look up the path
`/net/terra/export/ws/usr/src/uts`
2. The autofs mount at `/net` intercepts the lookup of `terra` and passes it on to the automountd as a mount request.
3. The automountd looks up `terra` in the hosts map and finds through the mount protocol that `terra` exports `"/usr"`, and `"/export/ws"`. Thus two mounts are required on the client.

4. For each mount the daemon performs a `mkdir` to create the required mountpoint and invokes the NFS `mount` program to perform the mount, e.g. (underscore represents a trailing space)

```
mkdir /net/terra/usr_  
mount /net/terra/usr_  
mkdir /net/terra/export/ws_  
mount /net/terra/export/ws_
```

Any process that uses a path that passes through `/net/terra` without a trailing space will be blocked by the autofs at `/net/terra` until the mounts are complete.

5. The daemon responds to the autofs that the mounts were successful.
6. The autofs unblocks the process(es) at `/net/terra` and allows lookups to proceed into the new mounts.

With these "special" paths, blockage avoidance is inherited from the daemon by a delegated mount or unmount process. It is also a capability that is unique to each thread within the daemon if deadlock between threads via access to other autofs mountpoints is to be avoided. A similar process occurs for unmounting described in section 3.3.

Although this solution to the deadlock problem is inelegant, in practice it works very well. Space terminated paths are not well supported by Unix and are generally not used. The space terminated path is significant only while a mounting operation is in progress.

2.8. Performance Improvement

Since it takes time to perform a mount, the first access to any automounted filesystem will take slightly longer if the filesystem is not already mounted. NFS-based automounters impose an additional naming overhead while the system is mounted since the kernel must request a symbolic link from the automountd daemon to locate the filesystem in the `/tmp_mnt` directory. This daemon involvement through symbolic links incurs an overhead in kernel/user context switching and data movement. Since the autofs automounter does its mounts in-place, the daemon is involved only when a filesystem needs to be mounted. Once it is mounted paths to the filesystem can be evaluated entirely within the kernel – the daemon is not required. This provides a performance improvement for naming operations.

Evaluating a `stat` system call 10,000 times with a path that traverses an automounter mountpoint shows a significant improvement in elapsed time.

Automounter	Clock Time (sec)	System Time (sec)
NFS based	32.9	17.0
Autofs based	1.8	1.5

3.0 The Automountd

The autofs is a very simple filesystem. It exists only to catch references to filesystems and provide a directory structure in which mounts can be made. The bulk of the automounter implementation remains in the automount daemon. This section describes the features of the automount daemon.

3.1. The RPC Protocol

The autofs uses a simple RPC protocol to communicate with the automountd. The protocol has two procedures: `mount` – to look up a name and perform mounts, and `unmount` – to request a set of filesystems be unmounted.

The `mount` procedure conveys the following information to the daemon:

1. The name to be looked up. To the daemon this is just a simple string. In the case of a direct mount the string is the pathname of the autofs mountpoint e.g. `/usr/man`. For an indirect mount it is the name of a directory entry e.g. `"brent"`.
2. The map name, e.g. `"auto_home"`.

3. Default mount options. A string of mount options that will be used for the mount – unless overridden by mount options provided in the map entry.
4. The path of the autofs mount, e.g. “/home”.

The response to this call is just an integer. If zero it indicates success (one or more mounts were done) – otherwise failure and the value is an `errno` (no such name in map, or mount unsuccessful).

The `unmount` call conveys a linked list of device identifiers to the daemon. These correspond to filesystems to be unmounted. Unmount also responds with a simple integer to indicate success or failure.

3.2. Statelessness

The daemon requires no preexisting information in order to service a mount or unmount request. All the information it needs is contained in the mount or unmount request from the kernel. A stateless daemon can be killed/restarted with no net loss in quality of service. This was a useful feature during development when we frequently killed the daemon and started a new version. Another feature of this statelessness is that the daemon need not be notified of changes in autofs mountpoints or maps.

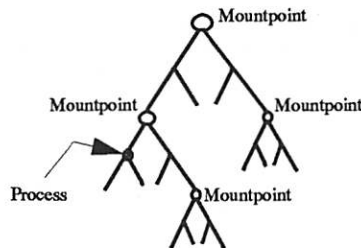
3.3. Auto Unmounting

To avoid an ever-increasing number of mounts, periodic unmounting is required. An unmount thread in the autofs checks the autofs mounts every 5 minutes. Filesystems that have not been referenced for 5 minutes are candidates for unmounting. Since there is no reliable way of knowing whether a filesystem is busy, the unmount thread sends an unmount request to the automountd with a list of the filesystem device identifiers that correspond to filesystems comprising a mounted hierarchy. The daemon attempts to unmount each of the filesystems and if any unmount fails because the filesystem is busy, the daemon remounts any prior successfully unmounted filesystems and responds with a failure notification to the unmount thread in the kernel. The unmount thread will retry the unmount later.

3.4. Safe Unmounting

The automounter has the ability to treat multiple mounts as a single unit. Several filesystems can be mounted and unmounted together. The mounts may be hierarchical.

Figure 8. Mount hierarchies are problematic for unmounting. A process may have a file open within some filesystem within the hierarchy that may keep the filesystem “busy” and not unmountable. If an unmount attempt succeeds elsewhere in the hierarchy, it’s mountpoint will appear to the process to be empty – until the automounter discovers the busy filesystem and remounts the filesystems that successfully unmounted.



To unmount a hierarchy of mounts the automounter starts at the deepest mounts in the hierarchy and proceeds upward toward the root – since a filesystem with a sub-mount cannot be unmounted until the sub-mount is first unmounted. If any of the unmounts fails the automounter is required to remount any sub-mounts that succeeded and restore the hierarchy to its original state. This procedure is unsafe because a process working within the hierarchy may choose an inopportune moment to reference one of the filesystems that is temporarily unmounted and fail with a missing file error.

In a future implementation of the autofs we would like to add a new VFS operation that can be used to check whether the filesystem is busy – without attempting an unmount. At the root of each mount hierarchy it must first block access to lookups to prevent processes from entering a hierarchy during the activity check. After determining which filesystems comprise the hierarchy, the unmount thread would call `VFS_BUSY` to obtain a count of active references to the filesystem. If the reference count is equal to the number of immediate sub-mounts then the filesystem could be considered to be “not busy”. If all filesystems in the hierarchy were “not busy” the autofs could send an unmount message to the daemon containing the filesystem identifiers for the filesystems to be unmounted. Since there is a guarantee that there are no open files or active processes in the hierarchy, this unmounting procedure would be safe.

3.5. Local Mounts

The old automounter took the opportunity to “cheat” and point its symbolic links at existing mounts whenever it could. If, in a map entry, the automounter noticed that the server name was the same as its local host, then it would point its symbolic link at the indicated local directory.

Since an autofs-based automounter doesn’t use symbolic links, it needs to reference local directories some other way. The automountd uses a loopback mount (lofs) to do this. Unlike a symbolic link loopback mounts are much less visible and can be evaluated more quickly than a symbolic link.

3.6. Filesystem Independence

The old automounter could mount only NFS filesystems. The autofs automounter can mount any kind of filesystem. Two changes to the map syntax were required to support this:

1. A new mount option `fstype` was provided to allow the filesystem type to be given. If the option is not given then “nfs” is assumed as the filesystem type.
2. The previously the map syntax now assumed that mount information took the NFS-specific form `server:/path`. Filesystem type-specific information can now be any string, though due to avoid map parser ambiguity a string that begins with a slash must be escaped with a colon to distinguish it from a mountpoint specification, e.g.

```
cdrom -fstype=hsfs,ro :/dev/sr0
```

This provides the automounter with enough information to invoke the filesystem-specific mount command e.g. given an `fstype` of “hsfs” the automounter will execute the command “`/usr/lib/fs/hsfs/mount`” passing the mount options (with “`fstype=hsfs`” removed) and the device path following the colon.

3.7. Executable Maps

Given a name, the automounter locates the required mount information by lookup in a name service (NIS or NIS+) or by searching an ASCII file. A more dynamic name-to-location mapping is provided by an executable map. If the map is a local file and has its execute bit set, then the automounter will `popen` the file (execute it) and provide the name to be looked up as an argument. The output on `stdout` is taken as the content of the map entry.

The following executable map duplicates the functionality of the `-hosts` map that we normally use under `/net`. It takes a server name as an argument and returns a map entry that mounts all the exported filesystems from the server.

```
#!/bin/sh
SERVER=$1

# If it's our own host just do a
# loopback mount of the root

if [ `uname -n` = $SERVER ]; then
    echo "-fstype=lofs :/"
    exit
fi

# Get the server's export list, turn it into
# a map entry, and sort it so the mounts are
# done in the right order

showmount -e $SERVER |
awk 'NR > 1 {print $1"\t"$SERVER:"$1 " \\"}' |
sort
```

The next script provides a name space that can be used to locate an entry in any NIS automounter map. It takes a name of the form key.mapname e.g. /nis/brent.auto.home is a path to my home directory.

```
#!/bin/ksh
# Parses names of the form key.nismap

KEY=${1%%.*}
MAP=${1#*.}
/usr/bin/ypmatch $KEY $MAP |
sed "s/&/$KEY/g"
```

The sed substitution for "&" is required here to substitute the parsed key since the automounter's substitution would insert the entire key used for the lookup.

4.0 Hierarchical Autofs Mounts

Since the autofs daemon can mount any type of filesystem, an automount map may contain entries that mount autofs filesystems. The maps that correspond to these autofs mounts may contain entries that refer to additional autofs mounts. This is best explained with an example. Assume that an organization comprising many departments wishes to organize a shared namespace under a "/org" directory that is visible to all users. Each department separately administers its own automounter map for its portion of the namespace. The master map needs just a single entry for /org.

```
# Directory      Map Name
/org             auto_org
```

The auto_org map looks like this:

```
finance      -fstype=autofs      auto_finance
marketing    -fstype=autofs      auto_marketing
legal        -fstype=autofs      auto_legal
research     -fstype=research    auto_research
eng          -fstype=autofs      auto_eng
```

and the engineering department's map "auto_eng" looks like this:

```
releases      bigiron:/export/releases
tools         mickey,minnie:/export/tools
source        -fstype=autofs      auto_eng_source
projects      -fstype=autofs      auto_eng_projects
```

This map structure would be of interest only to administrators. A user interested in the "blackhole" project within engineering might use the path:

```
/org/eng/projects/blackhole
```

Beginning with the autofs mount at /org, the evaluation of this path would dynamically create additional autofs mounts at /org/eng and /org/eng/projects. Since the autofs mounts are created only when needed, changes to maps require no action to become visible at the user's workstation. The automount command needs to be run only when changes are made to the master map, or to a direct map.

Hierarchical autofs mounts provide a framework within which large shared filesystem namespaces can be organized. Together with a name service like NIS+ [7] that supports information sharing across administrative domains, the maintenance of the shared namespace can be effectively decentralized.

5.0 Summary

The autofs provides the kernel support necessary for automatic mounting. It is easier to use, more efficient, more robust, and more flexible than the current user-level NFS server implementation. The autofs automounter is completely compatible with existing automounter maps.

Acknowledgments

Chris Silveri is responsible for the concept of a kernel-based filesystem to support automatic mounting.

References

- [1] B.N. Bershad and C.B. Pinkerton, "Watchdogs - Extending the UNIX File System", Winter 1988 Usenix Conference Proceedings.
- [2] Brent Callaghan, Tom Lyon. "The Automounter", Winter 1989 Usenix Conference Proceedings.
- [3] Brent Callaghan. "The Automounter - Using it Effectively", 1990 Sun User Group Conference Proceedings.
- [4] Brent Callaghan, "The Automounter - Solaris 2.0 and Beyond", 1992 Sun User Group Conference Proceedings.
- [5] Jan-Simon Pendry, "Amd - An Automounter", Technical Report, Department of Computing, Imperial College, London, England, 1989.
- [6] Ron Minnich, "The AutoCacher: A File Cache Which Operates at the NFS Level", Winter 1993 Usenix Conference Proceedings.
- [7] Chuck McManis, "Naming Systems: A Replacement for NIS", September 1991 Sun UK User Group Conference Proceedings.
- [8] S.R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX", Summer 1986 Usenix Conference Proceedings.

Author Information

Before coming to SunSoft, Brent Callaghan worked for 6 years as a system programmer at the University of Auckland, New Zealand, and two years as a contractor at AT&T Information Systems in Lincroft, New Jersey. For 6 years he has worked on a variety of projects within the NFS group at Sun with an emphasis on automounting. His E-mail address is brent@eng.sun.com.

Satinder Singh is a Member of Technical Staff at Sun Microsystems. He holds a Bachelors degree in mechanical Engineering from Banaras Hindu University, India and a Masters in Computer Science from Stanford University. His E-mail address is satinder@eng.sun.com.

Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing

*Erez Zadok and Dan Duchamp
Computer Science Department
Columbia University*

ABSTRACT

We describe a mechanism for replacing files, including open files, of a read-only file system while the file system remains mounted; the act of replacement is transparent to the user. Such a “hot replacement” mechanism can improve fault-tolerance, performance, or both. Our mechanism monitors, from the client side, the latency of operations directed at each file system. When latency degrades, the client automatically seeks a replacement file system that is equivalent to but hopefully faster than the current file system. The files in the replacement file system then take the place of those in the current file system. This work has particular relevance to mobile computers, which in some cases might move over a wide area. Wide area movement can be expected to lead to highly variable response time, and give rise to three sorts of problems: increased latency, increased failures, and decreased scalability. If a mobile client moves through regions having partial replicas of common file systems, then the mobile client can depend on our mechanism to provide increased fault tolerance and more uniform performance.

1 Introduction

The strongest trend in the computer industry today is the miniaturization of workstations into portable “notebook” or “palmtop” computers. Wireless network links [3] and new internetworking technology [8] offer the possibility that computing sessions could run without interruption even as computers move, using information services drawn from an infrastructure of (mostly) stationary servers.

We contend that operation of mobile computers according to such a model will raise problems that require re-thinking certain issues in file system design.¹ One such issue is how to cope with a client that moves regularly yet unpredictably over a wide area.

Several problems arise when a client moves a substantial distance away from its current set of servers. One is worse latency, since files not cached at the client must be fetched over longer distances. Another problem is increased probability of loss of connectivity, since gateway failures often lead to partitions. The final problem is decreased overall system “scalability:” more clients moving more data over more gateways means greater stress on the shared network.

One obvious way to mitigate these problems is to ensure that a file service client uses “nearby” servers at all times. A simple motivating example is that if a computer moves from New York to Boston, then in many cases it is advantageous to switch to using the Boston copies of “common” files like those in `/usr/ucb`. As the client moves, the file service must be able to provide service first from one server, then from another. This switching mechanism should require no action on the

¹Examples of such re-thinking can be found in [25] and [26].

part of administrators (since presumably too many clients will move too often and too quickly for administrators to track conveniently) and should be invisible to users, so that users need not become system administrators.

We have designed and implemented just such a file system — it adaptively discovers and mounts a “better” copy of a read-only file system which is fully or partially replicated. We define a better replica to be one providing better latency. Running our file service gives a mobile client some recourse to the disadvantages mentioned above. Our mechanism monitors file service latencies and, when response becomes inadequate, performs a dynamic attribute-guided search for a suitable replacement file system.

Many useful “system” file systems — and almost all file systems that one would expect to be replicated over a wide area — are typically exported read-only. Examples include common executables, manual pages, fonts, include files, etc. Indeed, read-only areas of the file space are growing fast, as programs increase the amount of configuration information, images, and on-line help facilities.

Although our work is motivated by the perceived needs of mobile computers that might roam over a wide area and/or frequently cross between public and private networks, our work can be useful in any environment characterized by highly variable response time and/or high failure rates.

Note that for a client to continue use of a file system as it moves, there must be underlying network support that permits the movement of a computer from one network to another without interruption of its sessions. Several such schemes have been developed [8, 7, 28, 29].

The remainder of this paper is organized as follows. In order to make a self-contained presentation, Section 2 provides brief explanations of other systems that we use in constructing ours. Section 3 outlines our design and Section 4 evaluates the work. Lastly, we mention related work in Section 5 and summarize in Section 6.

2 Background

Our work is implemented in and on SunOS 4.1.2. We have changed the kernel’s client-side NFS implementation, and outside the operating system we have made use of the Amd automounter and the RLP resource location protocol. Each is explained briefly below.

2.1 NFS

Particulars about the NFS protocol and implementation are widely known and published [20, 12, 9, 19]. For the purpose of our presentation, the only uncommon facts that need to be known are:

- Translation of a file path name to a vnode is done mostly within a single procedure, called `au_lookuppn()`, that is responsible for detecting and expanding symbolic links and for detecting and crossing mount points.
- The name of the procedure in which an NFS client makes RPCs to a server is `rfscall()`.

We have made substantial alterations to `au_lookuppn()`, and slight alterations to `rfscall()`, `nfs_mount()`, `nfs_unmount()` and `copen()`.² We added two new system calls: one for controlling and querying the added structures in the kernel, and the other for debugging. Finally, we added fields to three major kernel data structures: vnode and vfs structures and the open file table.

2.2 RLP

We use the RLP resource location protocol [1] when seeking a replacement file system. RLP is a general-purpose protocol that allows a site to send broadcast or unicast request messages asking either of two questions:

²`Copen()` is the common code for `open()` and `create()`.

1. Do you (recipient site) provide this service?
2. Do you (recipient site) know of any site that provides this service?

A service is named by the combination of its transport service (e.g., TCP), its well-known port number as listed in `/etc/services`, and an arbitrary string that has meaning to the service. Since we search for an NFS-mountable file system, our RLP request messages contain information such as the NFS transport protocol (UDP [16]), port number (2049) and service-specific information such as the name of the root of the file system.

2.3 Amd

Amd [15] is a widely-used automounter daemon. Its most common use is to demand-mount file systems and later unmount them after a period of disuse; however, Amd has many other capabilities.

Amd operates by mimicking an NFS server. An Amd process is identified to the kernel as the “NFS server” for a particular mount point. The only NFS calls for which Amd provides an implementation are those that perform name binding: `lookup`, `readdir`, and `readlink`. Since a file must have its name resolved before it can be used, Amd is assured of receiving control during the first use of any file below an Amd mount point. Amd checks whether the file system mapped to that mount point is currently mounted; if not, Amd mounts it, makes a symbolic link to the mount point, and returns to the kernel. If the file system is already mounted, Amd returns immediately.

An example, taken from our environment, of Amd’s operation is the following. Suppose `/u` is designated as the directory in which all user file systems live; Amd services this directory. At startup time, Amd is instructed that the mount point is `/n`. If any of the three name binding operations mentioned above occurs for any file below `/u`, then Amd is invoked. Amd consults its maps, which indicate that `/u/foo` is available on server `bar`. This file system is then mounted locally at `/n/bar/u/foo` and `/u/foo` is made a symbolic link to `/n/bar/u/foo`. (Placing the server name in the name of the mount point is purely a configuration decision, and is not essential.)

Our work is not dependent on Amd; we use it for convenience. Amd typically controls the (un)mounting of all file systems on the client machines on which it runs, and there is no advantage to our work in circumventing it and performing our own (un)mounts.

2.3.1 How Our Work Goes Beyond Amd

Amd does not already possess the capabilities we need, nor is our work a simple extension to Amd. Our work adds at least three major capabilities:

1. Amd keeps a description of where to find to-be-mounted file systems in “mount-maps.” These maps are written by administrators and are static in the sense that Amd has no ability for automated, adaptive, unplanned discovery and selection of a replacement file system.
2. Because it is only a user-level automount daemon, Amd has limited means to monitor the response of `rfscall()` or any other kernel routine.
Many systems provide a tool, like `nfsstat`, that returns timing information gathered by the kernel. However, `nfsstat` is inadequate because it is not as accurate as our measurements, and provides weighted average response time rather than measured response time. Our method additionally is less sensitive to outliers measures both short-term and long-term performance.
3. Our mechanism provides for transparently switching *open* files from one file system to its replacement.

3 Design

The key issues we see in this work are:

1. Is a switching mechanism really needed? Why not use the same file systems no matter where you are?
2. When and how to switch from one replica to another.
3. How to ensure that the new file system is an acceptable replacement for the old one.
4. How to ensure consistency if updates are applied across different replicas.
5. Fault tolerance: how to protect a client from server unavailability.
6. Security: NFS is designed for a local "workgroup" environment in which the space of user IDs is centrally controlled.

These issues are addressed below.

3.1 Demonstrating the Need

We contend that adaptive client-server matchups are desirable because running file system operations over many network hops is bad for performance in three ways: increased latency, increased failures, and decreased scalability. It is hard to ascertain exact failure rates and load on shared resources without undertaking a full-scale network study; however, we were able to gather some key data to support our claim. We performed a simple study to measure how latency increases with distance.

First, we used the *traceroute* program³ to gather $\langle \text{hop-count}, \text{latency} \rangle$ data points measured between a host at Columbia and several other hosts around the campus, city, region, and continent. Latencies were measured by a Columbia host, which is a Sun-4/75 equipped with a microsecond resolution clock. The cost of entering the kernel and reading the clock is negligible, and so the measurements are accurate to a small fraction of a millisecond.

Next, we mounted NFS file systems that are exported Internet-wide by certain hosts. We measured the time needed to copy 1MB from these hosts using a 1KB block size. A typical result is plotted in Figure 1. Latency jumps by almost two orders of magnitude at the tenth hop, which represents the first host outside Columbia.

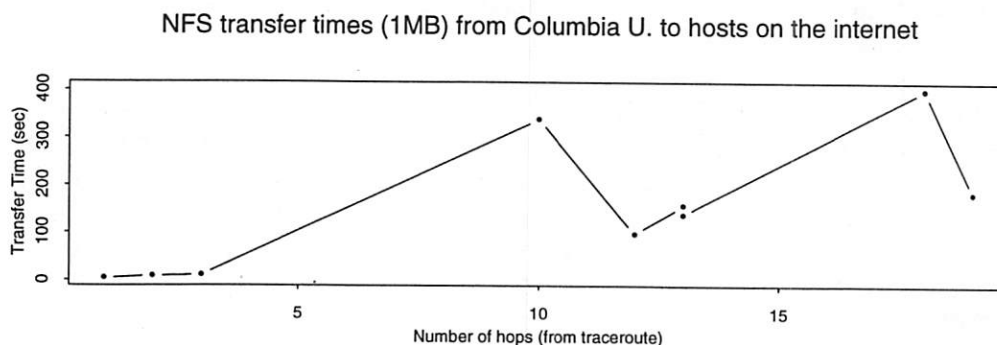


Figure 1: NFS Read Latency vs. Network Hop Count

³Written by Van Jacobson and widely available by anonymous ftp.

3.2 When to Switch

We have modified the kernel so that `rfscall()` measures the latency of every NFS `lookup` and maintains a per-filesystem data structure storing a number of recently measured latencies.

We chose to time the `lookup` operation rather than any other operation or mixture of operations for two reasons. The first is that `lookup` is the most frequently invoked NFS operation. We felt other calls would not generate enough data points to accurately characterize latency. The second reason is that `lookup` exhibits the least performance variability of the common NFS operations. Limiting variability of measured server latencies is important in our work, since we want to distinguish transient changes in server performance from long-term changes.

(At the outset of our work, we measured variances in the latency of the most common NFS operations and discovered huge swings, shown in Figure 2, even in an extended LAN environment that has been engineered to be uniform and not to have obvious bottlenecks. The measured standard deviations were 1027 msec for all NFS operations, 2547 msec for `read`, and 596 msec for `lookup`.)

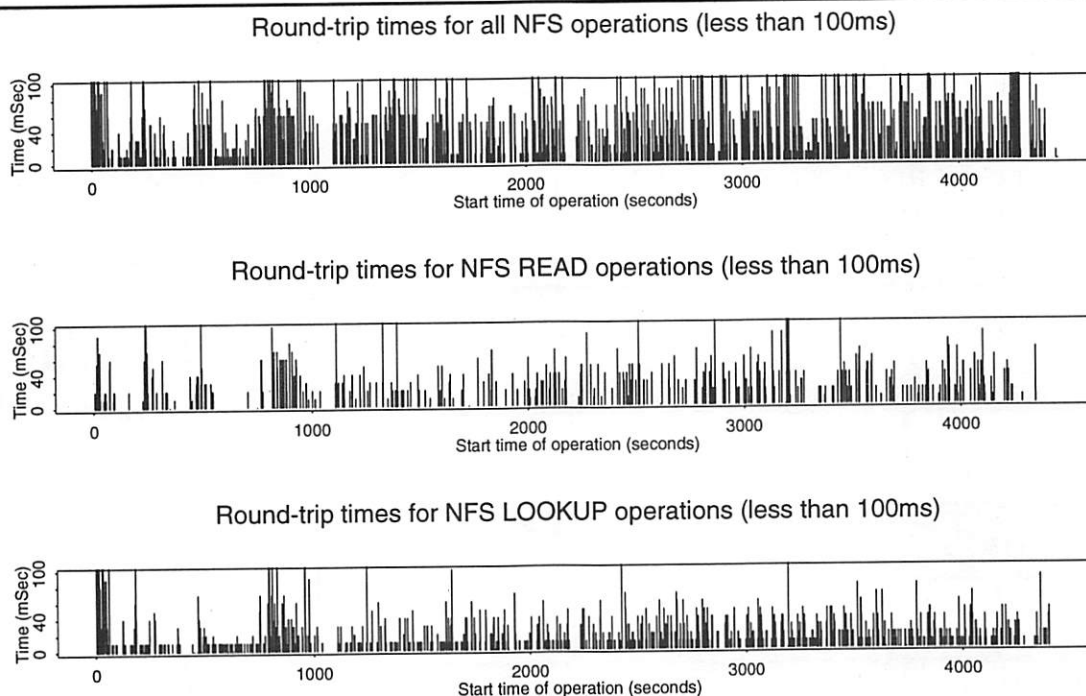


Figure 2: Variability and Latency of NFS operations

After addition of each newly measured `lookup` operation, the median latency is computed over the last 30 and 300 calls. We compute medians because medians are relatively insensitive to outliers. We take a data point no more than once per second, so during busy times these sampling intervals correspond to 30 seconds and 5 minutes, respectively. This policy provides insurance against anomalies like ping-pong switching between a pair of file systems: a file system can be replaced no more frequently than every 5 minutes.

The signal to switch is when, at any moment, the short-term median latency exceeds the long-term median latency by a factor of 2. Looking for a factor of two difference between short-term and long-term medians is our attempt to detect a change in performance which is substantial and “sudden,” yet not transient. The length of the short-term and long-term medians as well as the ratio

used to signal a switch are heuristics chosen after experimentation in our environment. All these parameters can be changed from user level through a debugging system call that we have added.

3.3 Locating a Replacement

When a switch is triggered, `rfscall()` starts a non-blocking RPC out to our user-level process that performs replacement, `nfsmgrd`.⁴ The call names the guilty file server, the root of the file system being sought, the kernel architecture, and any mount options affecting the file system. `Nfsmgrd` uses these pieces of information to compose and broadcast an RLP request. The file system name keys the search, while the server name is a filter: the search must not return the same file server that is already in use.

The RLP message is received by the `nfsmgrd` at other sites on the same broadcast subnet. To formulate a proper response, an `nfsmgrd` must have a view of mountable file systems stored at its site and also mounted file systems that it is using — either type could be what is being searched for. Both pieces of information are trivially accessible through `/etc/fstab`, `/etc/exports`, and `/etc/mtab`.

The `nfsmgrd` at the site that originated the search uses the first response it gets; we suppose that the speed with which a server responds to the RLP request gives a hint about its future performance. (The Sun Automounter [2] makes the same assumption about replicated file servers.) If a read-only replacement file system is available, `nfsmgrd` instructs `Amd` to mount it and terminates the out-of-kernel RPC, telling the kernel the names of the replacement server and file system. The flow of control is depicted in Figure 3.

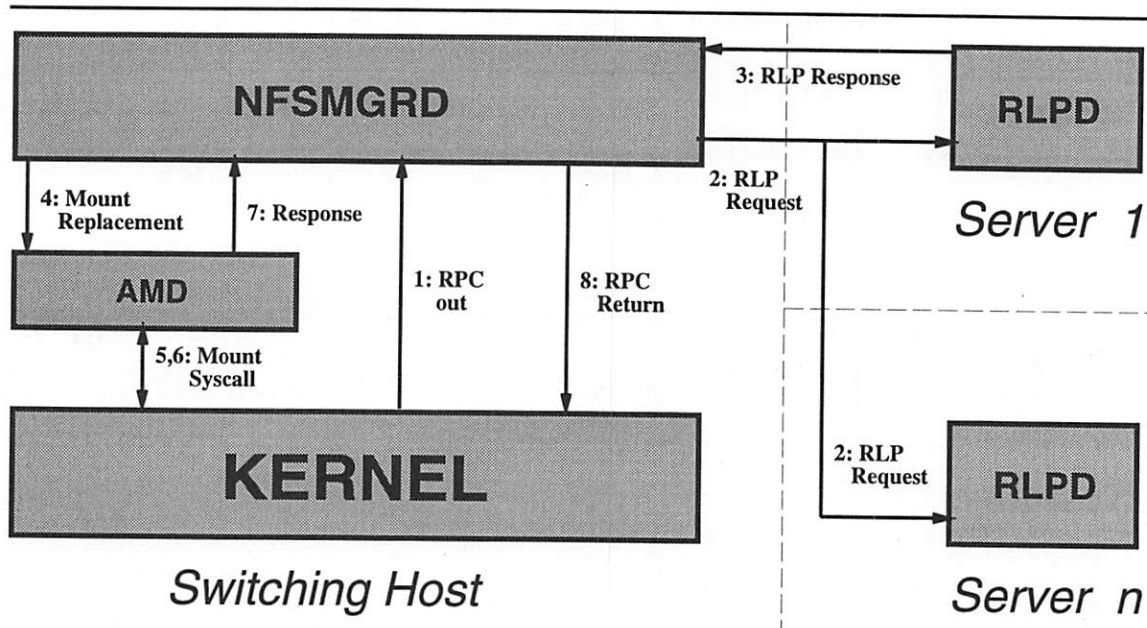


Figure 3: Flow of Control During a Switch

⁴Non-blocking operation is provided by a special kernel implementation of Sun RPC.

3.4 Using the Replacement

Once a replacement file system has been located and mounted, all future attempts to open files on the replaced file system will be routed to the replacement whenever they can be. Also, in all cases for which it is possible, *open files* on the replaced file system will be switched to their equivalents on the replacement. We describe these two cases in Sections 3.4.2 and 3.4.3, respectively.

3.4.1 Relevant Changes to Kernel Data Structures

In order to accommodate file system replacement, we have added some fields to three important kernel data structures: **struct vfs**, which describes mounted file systems; **struct vnode**, which describes open files; and **struct file**, which describes file descriptors.

The fields added to **struct vfs**, excluding debugging fields, are:

- The field **vfs_replaces** is valid in the **vfs** structure of the replacement file system; it points to the **vfs** structure of the file system being replaced.
- The field **vfs_replaced_by** is valid in the replaced file system's **vfs** struct; it points to the **vfs** structure of the replacement file system.
When a replacement file system is mounted, our altered version of **nfs_mount()** sets the replaced and replacement file systems pointing to each other.
- The field **vfs_nfsmgr_flags** is valid for any NFS file system. One flag indicates whether the file system is managed by *nfsmgrd*; another indicates whether a file system switch is in progress.
- The field **vfs_median_info** contains almost all of the pertinent information about the performance of the file system, including the 300 most recent **nfs_lookup()** response times.
- The field **vfs_dft** is the *Duplicate File Table* (DFT). This per-filesystem table lists which files in the replacement file system have been compared to the corresponding file on the original file system mounted by Amd. Only equivalent files can be accessed on the replacement file system. The mechanism for making comparisons is described in Section 3.4.2.
The size of the DFT is fixed (but changeable) so that new entries inserted will automatically purge old ones. This is a simple method to maintain "freshness" of entries.
The DFT is a hash table whose entries contain a file pathname relative to the mount point, a pointer to the **vfs** structure of the replacement file system, and an extra pointer for threading the entries in insertion order. This data structure permits fast lookups keyed by pathname and quick purging of older entries.

The only field added to **struct vnode** is **v_last_used**, which contains the last time that **rfscall()** made a remote call on behalf of this **vnode**. This information is used in "hot replacement," as described in Section 3.4.3.

The only field added to **struct file** is **f_path**, which contains the relative pathname from the mount point to the file for which the descriptor was opened. Different entries may have different pathnames for the same file if several hard links point to the file.

3.4.2 After Replacement: Handling New Opens

When Amd mounts a file system it makes a symlink from the desired location of the file system to the mount point. For example, **/u/foo** would be a symlink pointing to the real mount point of **/n/bar/u/foo**; by our local convention, this would indicate that server **bar** exports **/u/foo**. Users and application programs know only the name **/u/foo**.

The information that **bar** exports a proper version of **/u/foo** is placed in Amd's mount-maps by system administrators who presumably ensure that the file system **bar:/u/foo** is a good version of whatever **/u/foo** should be. Therefore, we regard the information in the client's Amd mount-maps as authoritative, and consider any file system that the client might mount and place at **/u/foo**

as a correct and complete copy of the file system. We call this file system *the master copy*, and use it for comparison against the replacement file systems that our mechanism locates and mounts.

The new open algorithm is shown in Figure 4. After a replacement has been mounted, whenever name resolution must be performed for any file on the replaced file system, the file system's DFT is first searched for the relative pathname. If the DFT indicates that the replacement file system has an equivalent copy of the file, then that file is used.

```
open() {
    examine vfs_replaced_by field to see if there is a replacement file system;
    if (no replacement file system) {
        continue name resolution;
        return;
    }
    if (DFT entry doesn't exist) {
        create and begin DFT entry;
        call out to perform file comparison;
        finish DFT entry;
    }
    if (files equivalent) {
        get vfs of replacement from vfs_replaces field;
        continue name resolution on replacement file system;
    } else
        continue name resolution on master copy;
}
```

Figure 4: New Open Algorithm

If the DFT contains an entry for the pathname, then the file on the replacement file system has already been compared to its counterpart on the master copy. A field in the DFT tells if the comparison was successful or not. If not, then the rest of the pathname has to be resolved on the master copy. If the comparison was successful, then the file on the replacement file system is used; in that case, name resolution continues at the root of the replacement file system.

If the DFT contains no entry for the pathname, then it is unknown whether the file on the replacement file system is equivalent to the corresponding file on the master copy.

To test equivalence, `au_lookupn()` calls out of the kernel to `nfsmgrd`, passing it the two host names, the name of the file system, and the relative pathname to be compared. A partial DFT entry is constructed, and a flag in it is turned on to indicate that there is a comparison in progress and that no other process should initiate the same comparison.⁵

`Nfsmgrd` then applies, at user level, whatever tests might be appropriate to determine whether the two files are equivalent. This flow of control is depicted in Figure 5. Presently, we are performing file checksum comparison: `nfsmgrd` calls a `checksumd` daemon on each of the file servers, requesting the checksum of the file being compared. `Checksumd`, which we have written for this work, computes MD4 [18] file checksums on demand and then stores them for later use; checksums can also be pre-computed and stored.

`Nfsmgrd` collects the two checksums, compares them, and responds to the kernel, telling `au_lookupn()` which pathname to use, always indicating the file on the replacement file system if possible. `Au_lookupn()` completes the construction of the DFT entry, unlocks it, and marks which

⁵This avoids the need to lock the call out to `nfsmgrd`.

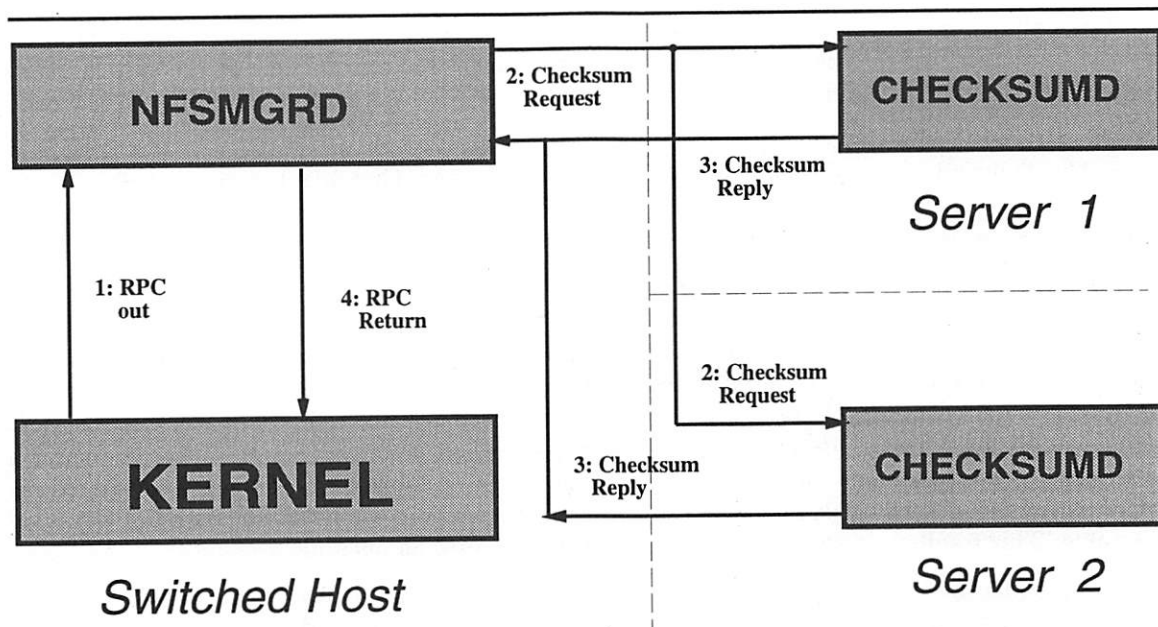


Figure 5: Flow of Control During File Comparison

vfs is the proper one to use whenever the same pathname is resolved again.

In this fashion, all new pathname resolutions are re-directed to the replacement file system whenever possible.

Note that the master copy could be unmounted (e.g., Amd by default unmounts a file system after a few minutes of inactivity), and this would not affect our mechanism. The next use of a file in that file system would cause some master copy to be automounted, before any of our code is encountered.

3.4.3 After Replacement: Handling Files Already Open

When a file system is replaced, it is possible that some files will be open on the replaced file system at the moment when the replacement is mounted. Were the processes with these open files to continue to use the replaced file system, several negative consequences might ensue. First, since the replacement is presumed to provide faster response, the processes using files open on the replaced file systems experience worse service. Second, since the total number of mounted file systems grows as replacements happen, the probability rises that some file system eventually becomes unavailable and causes processes to block. Further, the incremental effect of each successive file system replacement operation is reduced somewhat, since files that are open long-term do not benefit from replacement. Finally, kernel data structures grow larger as the number of mounted file systems climbs. Motivated by these reasons, we decided to switch *open files* from the replaced file system to the replacement file system whenever the file on the replacement file system is equivalent to that on the master copy.

Although this idea might at first seem preposterous, it is not, since we restrict ourselves to read-only file systems. We assume that files on read-only file systems⁶ change very infrequently and/or are updated with care to guard against inconsistent reads.⁷ Whether operating conditions uphold this assumption or not, the problem of a file being updated⁸ while being read exists independently of our work, and our work does not increase the danger.

⁶That is, they are *exported* as read-only to some hosts, although they might be exported as read-write to others.

⁷An example of "careful update" is provided by the SUP utility [22].

⁸That is, updated by a host to which the file system is exported read-write.

We allow for a replacement file system to be itself replaced. This raises the possibility of creating a "chain" of replacement file systems. Switching vnodes from the old file system to its replacement limits this chain to length two (the master copy and the current replacement) in steady state.

The "hot replacement" code scans through the global open file table, keying on entries by vfs. Once an entry is found that uses the file system being replaced, a secondary scan locates all other entries using the same vnode. In a single entry into the kernel (i.e., "atomically"), all file descriptors pointing to that vnode are switched, thereby avoiding complex questions of locking and reference counting.

Hot replacement requires knowing pathnames. Thanks to our changes, the vfs structure records the pathname it is mounted on and identifies the replacement file system; also, the relative pathname of the file is stored in the file table entry. This information is extracted, combined with the host names, and passed out to *nfsmgrd* to perform comparison, as described above. If the comparison is successful, the pathname on the replacement file system is looked up, yielding a vnode on the replacement file system. This vnode simply replaces the previous vnode in all entries in the open file table. This results in a switch the next time a process uses an open file descriptor.

Hot replacement is enabled by the statelessness of NFS and by the vfs/vnode interfaces within the kernel. Since the replaced server keeps no state about the client, and since the open file table knows only a pointer to a vnode, switching this pointer in every file table entry suffices to do hot replacement.

An interesting issue is at which time to perform the hot replacement of vnodes. Since each file requires a comparison to determine equivalence, switching vnodes of all the open files of a given file system could be a lengthy process. The four options we considered are:

1. Switch as soon as a replacement file system is mounted (the early approach).
2. Switch only if/when an RPC for that vnode hangs (the late approach).
3. Switch if/when the vnode is next used (the "on-demand" approach).
4. Switch whenever a daemon instructs it to (the "flexible" approach).

The decision to switch earlier or later is affected by the tradeoff that early switching more quickly switches files to the faster file system and improves fault tolerance by reducing the number of file systems in use, but possibly wastes effort. Vnode switching is a waste in all cases when a vnode exists that will not be used again. Early switching also has the disadvantage of placing the entire delay of switching onto the single file reference that is unlucky enough to be the next one.

We chose the "flexible" approach of having a daemon make a system call into the kernel which then sweeps through the open file table and replaces some of the vnodes which can be replaced. We made this choice for three reasons. First, we lacked data indicating how long a vnode lingers after its final use. Second, we suspected that such data, if obtained, would not conclusively decide the question in favor of an early or late approach. Third, the daemon solution affords much more flexibility, including the possibility of more "intelligent" decisions such as making the switch during an idle period.

We emphasize that the system call into the kernel switches "some" of the vnodes, since it may be preferable to bound the delay imposed on the system by one of these calls. Two such bounding policies that we have investigated are, first, switching only *N* vnodes per call, and, second, switching only vnodes that have been accessed in the past *M* time units. Assuming that file access is bursty (a contention supported by statistics [14]), the latter policy reduces the amount of time wasted switching vnodes that will never be used again. We are currently using this policy of switching only recently used vnodes; this policy makes use of the `v_last_used` field that we added to the vnode structure.

3.5 Security

The NFS security model is the simple uid/gid borrowed from UNIX, and is appropriate only in a “workgroup” situation where there is a central administrative authority. Transporting a portable computer from one NFS user ID domain to another presents a security threat, since processes assigned user ID X in one domain can access exported files owned by user ID X in the second domain.

Accordingly, we have altered `rfscall()` so that every call to a replacement file system has its user ID and group ID both mapped to “nobody” (i.e., value -2). Therefore, only world-readable files on replacement file systems can be accessed.

3.6 Code Size

Counting blank lines, comments, and debugging support, we have written close to 11,000 lines of C. More than half is for user-level utilities: 1200 lines for the RLP library and daemon, 3200 for `nfsmgrd`, 700 lines for `checksumd`, and 1200 lines for a control utility (called `nfsmgr_ctl`). New kernel code totals 4000 lines, of which 800 are changes to SunOS, mostly in the NFS module. The remaining 3200 lines comprise the four modules we have added: 880 lines to deal with storing and computing medians; 780 lines are the “core NFS management” code, which performs file system switching, pathname storage and replacement, and out-of-kernel RPC; 540 lines to manage the DFT; and 1000 lines to support the `nfsmgr_ctl` system call.

The `nfsmgr_ctl` system call allows query and control over almost all data structures and parameters of the added facility. We chose a system call over a `kmem` program for security. This facility was used heavily during debugging; however, it is meant also for system administrators and other interested users who would like to change these “magic” variables to values more suitable for their circumstances.

4 Evaluation

This system is implemented and is receiving use on a limited number of machines.

The goal of this work is to improve overall file system performance — under certain circumstances, at least — and to improve it enough to justify the extra complexity. For this method to really work, it must have:

1. Low overhead latency measurement between switches.
2. A quick switch.
3. Low overhead access to the replacement after a switch.
4. No anomalies or instabilities, like ping-pong switching.
5. No process hangs due to server failures.
6. No security or administrative complications.

We have carried out several measurements aimed at evaluating how well our mechanism meets these goals.

The *overhead between switches* is that of performance monitoring. The added cost of timing every `rfscall()` we found too small to measure. The cost of computing medians could be significant, since we retain 300 values. But we implemented a fast incremental median algorithm that requires just a negligible fraction of the time in `nfs_lookup()`. The kernel data structures are not so negligible: retaining 300 latency measurements costs about 2KB per file system. The reason for the expansion is the extra pointers that must be maintained to make the incremental median algorithm work. The extra fields in the `struct vfs`, `struct vnode`, `struct file` are small, with the exception of the DFT, which is large. The current size of each (per-filesystem) DFT is 60 slots which occupy a total of 1KB-2KB on average.

Our measured *overall switch time* is approximately 3 sec. This is the time between the request for a new file system and when the new file system is mounted (messages 1-8 in Figure 3). Three seconds is comparable to the time needed in our facility to mount a file system whose location is already encoded in Amd's maps, suggesting that most of the time goes to the mount operation.

The *overhead after a switch* consists mostly of doing equivalence checks outside the kernel; the time to access the vfs of the replacement file system and DFT during `au_lookupn()` is immeasurably small. Only a few milliseconds are devoted to calling `checksumd`: 5-7 msec if the checksum is already computed. This call to `checksumd` is done once and need not be done again so long as a record of equivalence remains in the DFT.

A major issue is how long to cache DFT entries that indicate equivalence. Being stateless, NFS does not provide any sort of server-to-client cache invalidation information. Not caching at all ensures that files on the replacement file system are always equal to those on the master copy; but of course this defeats the purpose of using the replacement. We suppose that most publicly-exported read-only file systems have their contents changed rarely, and thus one should cache to the maximum extent. Accordingly, we manage the DFT cache by LRU.

As mentioned above, *switching instabilities* are all but eliminated by preventing switches more frequently than every 5 minutes.

4.1 Experience

4.1.1 What is Read-Only

Most of the files in our facility reside on read-only file systems. However, sometimes one can be surprised. For example, GNU *Emacs* is written to require a world-writable lock directory. In this directory *Emacs* writes files indicating which users have which files in use. The intent is to detect and prevent simultaneous modification of a file by different processes. A side effect is that the "system" directory in which *Emacs* is housed (at our installation, `/usr/local`) must be exported read-write.

Deployment of our file service spurred us to change *Emacs*. We wanted `/usr/local` to be read-only so that we could mount replacements dynamically. Also, at our facility there are several copies of `/usr/local` per subnet, which defeats *Emacs*' intention of using `/usr/local` as a universally shared location. We re-wrote *Emacs* to write its lock files in the user's home directory since (1) for security, our system administrators wish to have as few read-write system areas as possible and, (2) in our environment by far the likeliest scenario of simultaneous modification is between two sessions of the same user, rather than between users.

4.1.2 Suitability of Software Base

Kernel. The vfs and vnode interfaces in the kernel greatly simplified our work. The hot replacement, in particular, proved far easier than we had feared, thanks to the vnode interface. The special out-of-kernel RPC library also was a major help. Nevertheless, work such as ours makes painfully obvious the benefits of implementing file service out of the kernel. The length and difficulty of the edit-compile-debug cycle, and the primitive debugging tools available for the kernel were truly debilitating.

RLP. RLP was designed in 1983, when the evils of over-broadcasting were not as deeply appreciated as they are today and when there were few multicast implementations. Accordingly, RLP is specified as a broadcast protocol. A more up-to-date protocol would use multicast. The benefits would include causing much less waste (i.e., bothering hosts that lack an RLP daemon) and contacting many more RLP daemons. Not surprisingly, we encountered considerable resistance from our bridges and routers when trying to propagate an RLP request. A multicast RLP request would travel considerably farther.

NFS. NFS is ill-suited for "cold replacement" (i.e., new opens on a replacement file system)

caused by mobility, but is well suited for “hot replacement” because of its statelessness.

NFS’ lack of cache consistency callbacks has long been bemoaned, and it affects this work since there is no way to invalidate DFT entries. Since we restrict ourselves to read-only files, the danger is assumed to be limited, but is still present. Most newer file service designs include cache consistency protocols. However, such protocols are not necessarily a panacea. Too much interaction between client and server can harm performance, especially if these interactions take place over a long distance and/or a low bandwidth connection. See [27] for a design that can ensure consistency with relatively little client-server interaction.

The primary drawback of using NFS for mobile computing is its limited security model. Not only can a client from one domain access files in another domain that are made accessible to the same user ID number, but even a well-meaning client cannot prevent itself from doing so, since there is no good and easy way to tell when a computer has moved into another uid/gid domain.

5 Related Work

It is a thesis of our work that in order for mobile computing to become the new standard model of computing, adaptive resource location and management will have to become an automatic function of distributed services software. The notion of constantly-networked, portable computers running modern operating systems is relatively new. Accordingly, we know of no work other than our own (already cited) on the topic of adaptive, dynamic mounting.

The Coda file system [21] supposes that mobile computing will take place in the form of “disconnected operation,” and describes in [11] a method in which the user specifies how to “stash” (read/write) files before disconnection and then, upon reconnection, have the file service run an algorithm to detect version skew. Coda can be taken as a point of contrast to our system, since the idea of disconnection is antithetical to our philosophy. We believe trends in wireless communication point to the ability to be connected any time, anywhere. Users may *decide* not to connect (e.g., for cost reasons) but will not be *forced* not to connect (e.g., because the network is unreliable or not omnipresent). We call this mode of operation *elective connectivity*.

An obvious alternative to our NFS-based effort is to employ a file system designed for wide-area and/or multi-domain operation. Such file systems have the advantages of a cache consistency protocol and a security model that recognizes the existence of many administrative domains. Large scale file systems include AFS [6] and its spinoffs, Decorum [10] and IFS (Institutional File System) [5]. Experiments involving AFS as a “nation-wide” file service have been going on for years [23]. This effort has focused on stitching together distinct administrative domains so as to provide a single unified naming and protection space. However, some changes are needed to the present authentication model in order to support the possibility of a mobile client relocating in a new domain. In particular, if the relocated client will make use of local services, then there should be some means whereby one authentication agent (i.e., that in the new domain) would accept the word of another authentication agent (i.e., that in the client’s home domain) regarding the identity of the client.

The IFS project has also begun to investigate alterations to AFS in support of mobile computers [4]. Specifically, they are investigating cache pre-loading techniques for disconnected operation and transport protocols that are savvy about the delays caused by “cell handoff” — the time during which a mobile computer moves from one network to another.

Plan 9’s *bind* command has been designed to make it easy to mount new file systems. In particular, file systems can be mounted “before” or “after” file systems already mounted at the same point. The before/after concept replaces the notion of a search path. Plan 9 also supports the notion of a “union mount” [17]. The Plan 9 *bind* mechanism is a more elegant alternative to our double mounting plus comparison. However, a binding mechanism — even an unusually flexible one such as that of Plan 9 — addresses only part of the problem of switching between file systems. The harder part of the problem is determining *when* to switch and *what* to switch to.

6 Conclusion

We have described the operation, performance, and convenience of a transparent, adaptive mechanism for file system discovery and replacement. The adaptiveness of the method lies in the fact that a file service client no longer depends solely on a static description of where to find various file systems, but instead can invoke a resource location protocol to inspect the local area for file systems to replace the ones it already has mounted.

Such a mechanism is generally useful, but offers particularly important support for mobile computers which may experience drastic differences in response time as a result of their movement. Reasons for experiencing variable response include: (1) moving beyond the home administrative domain and so increasing the “network distance” between client and server and (2) moving between high-bandwidth private networks and low-bandwidth public networks (such movement might occur even within a small geographic area). While our work does not address how to access replicated read/write file systems or how to access one’s home directory while on the move, our technique does bear on the problems of the mobile user. Specifically, by using our technique, a mobile user can be relieved of the choice of either suffering with poor performance or devoting substantial local storage to “system” files.⁹ Instead, the user could rely on our mechanism to continuously locate copies of system files that provide superior latency, while allocating all or most of his/her limited local storage to caching or stashing read/write files such as those from the home directory.

Our work is partitioned into three modular pieces: heuristic methods for detecting performance degradation and triggering a search; a search technique coupled with a method for testing equivalence versus a master copy; and a method for force-switching open files from the use of vnodes on one file system to vnodes on another (i.e., “hot replacement”). There is little interrelationship among these techniques, and so our contributions can be viewed as consisting not just of the whole, but also of the pieces. Accordingly, we see the contributions of our work as:

1. The observation that file system switching might be needed and useful.
2. The idea of an automatically self-reconfiguring file service, and of basing the reconfiguration on measured performance.
3. Quantification of the heuristics for triggering a search for a replacement file system.
4. The realization that a “hot replacement” mechanism should not be difficult to implement in an NFS/vnodes setting, and the implementation of such a mechanism.

6.1 Future Work

There are several directions for future work in this area.

The major direction is to adapt these ideas to a file service that supports a more appropriate security model. One part of an “appropriate” security model is support for cross-domain authentication such that a party from one domain can relocate to another domain and become authenticated in that domain. Another part of an appropriate security model should include accounting protocols allowing third parties to advertise and monitor (i.e., “sell”) the use of their exported file systems. Within the limited context of NFS, a small step in the right direction would be a mechanism that allows clients (servers) to recognize servers (clients) from a different domain. The most recent version of Kerberos contains improved support for cross-domain authentication, so another step in the right direction would be to integrate the latest Kerberos with NFS, perhaps as originally sketched in [24].

Another desirable idea is to convert from using a single method of exact file comparison (i.e., *checksumd*) to per-user, possibly inexact comparison. For example, object files produced by *gcc*

⁹One might suppose that a “most common subset” of system files could be designated and loaded, and this is true. However, specifying such a subset is ever harder as programs depend on more and more files for configuration and auxiliary information. This approach also increases the user’s responsibility for system administration, which we regard as a poor way to design systems.

contain a timestamp in the first 16 bytes; two object files may be equal except for the embedded timestamps, which can be regarded as an insignificant difference. Another example is that data files may be equal except for gratuitous differences in floating-point format (e.g., 1.7 vs. 1.7000 vs. 1.70e01). Source files may be compared ignoring comments and/or white space. Intelligent comparison programs like *diff* or *spiff* [13] know how to discount certain simple differences.

Minor extensions to our work include: converting RLP from a broadcast protocol to a multicast protocol; and reimplementing in an environment (e.g., multi-server Mach 3.0) that supports out-of-kernel file service implementations.

7 Acknowledgements

We thank an anonymous member of the program committee for the suggestion to use file checksums. We thank the program committee, especially David Rosenthal and Matt Blaze, for valuable advice about presentation and emphasis.

This work was supported in part by the New York State Science and Technology Foundation's Center for Advanced Technology in Computers and Information Systems; by a National Science Foundation CISE Institutional Infrastructure grant, number CDA-90-24735; and by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

8 References

- [1] M. Accetta. Resource Location Protocol. RFC 887, IETF Network Working Group, December 1983.
- [2] B. Callaghan and T. Lyon. The Automounter. In *Proc. 1989 Winter USENIX Conf.*, pages 43–51, January 1989.
- [3] D. C. Cox. A Radio System Proposal for Widespread Low-power Tetherless Communication. *IEEE Trans. Communications*, 39(2):324–335, February 1991.
- [4] P. Honeyman. Taking a LITTLE WORK Along. CITI Report 91-5, Univ. of Michigan, August 1991.
- [5] J. Howe. Intermediate File Servers in a Distributed File System Environment. CITI Report 92-4, Univ. of Michigan, June 1992.
- [6] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Trans. Computer Systems*, 6(1):51–81, February 1988.
- [7] J. Ioannidis et al. Protocols for Supporting Mobile IP Hosts Draft RFC, IETF Mobile Hosts Working Group, June 1992.
- [8] J. Ioannidis, D. Duchamp and G. Q. Maguire Jr. IP-based Protocols for Mobile Internetworking. In *Proc. SIGCOMM '91*, pages 235–245. ACM, September 1991.
- [9] C. Juszczak. Improving the Performance and Correctness of an NFS Server. In *Proc. 1989 Winter USENIX Conf.*, pages 53–63, January 1989.
- [10] M. L. Kazar et al. Decorum File System Architectural Overview. In *Proc. 1990 Summer USENIX Conf.*, pages 151–163, June 1990.
- [11] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Computer Systems*, 10(1):3–25, February 1992.
- [12] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun Unix. In *Proc. 1986 Summer USENIX Conf.*, pages 238–247, June 1986.

- [13] D. Nachbar. Spiff – A Program for Making Controlled Approximate Comparisons of Files. In *Proc. 1986 Summer USENIX Conf.*, pages 238–247, June 1986.
- [14] J. Ousterhout et al. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 15–24, December 1985.
- [15] J. Pendry and N. Williams. *Amd - The 4.4 BSD Automounter*. Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.
- [16] J. Postel. User Datagram Protocol. RFC 768, IETF Network Working Group, August 1980.
- [17] D. Presotto et al. Plan 9, A Distributed System. In *Proc. Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.
- [18] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1186, IETF Network Working Group, April 1992.
- [19] D. S. H. Rosenthal. Evolving the Vnode Interface. In *Proc. 1990 Summer USENIX Conf.*, pages 107–117, June 1990.
- [20] R. Sandberg et al. Design and Implementation of the Sun Network Filesystem. In *Proc. 1985 Summer USENIX Conf.*, pages 119–130, June 1985.
- [21] M. Satyanarayanan et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.
- [22] S. Shafer and M. R. Thompson. The SUP Software Upgrade Protocol. Unpublished notes available by ftp from mach.cs.cmu.edu:/mach3/doc/unpublished/sup/sup.doc
- [23] A. Z. Spector and M. L. Kazar. Uniting File Systems. *UNIX Review*, 7(3):61–71, March 1989.
- [24] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. 1988 Winter USENIX Conf.*, pages 191–202, February, 1988.
- [25] C. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.
- [26] C. Tait and D. Duchamp. Service Interface and Replica Consistency Algorithm for Mobile File System Clients. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197. IEEE, December 1991.
- [27] C. Tait and D. Duchamp. An Efficient Variable Consistency Replicated File Service. In *File Systems Workshop*, pages 111–126. USENIX, May 1992.
- [28] F. Teraoka, Y. Yokote, and M. Tokoro. A network Architecture Providing Host Migration Transparency. In *Proc. SIGCOMM '91*, pages 209–220. ACM, September 1991.
- [29] H. Wada et al. Mobile Computing Environment Based on Internet Packet Forwarding. In *Proc. 1993 Winter USENIX Conf.*, pages 503–517, January 1993.

9 Author Information

Erez Zadok is an MS candidate and full-time Staff Associate in the Computer Science Department at Columbia University. This paper is a condensation of his Master's thesis. His primary interests include operating systems, file systems, and ways to ease system administration tasks. In May 1991, he received his B.S. in Computer Science from Columbia's School of Engineering and Applied Science. Erez came to the United States six years ago and has lived in New York "sin" City ever since. In his rare free time Erez is an amateur photographer, science fiction devotee, and rock-n-roll fan.

Mailing address: 500 West 120th street, Columbia University, New York, NY 10027. Email address: `ezk@cs.columbia.edu`.

Dan Duchamp is an Assistant Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he was recently named an Office of Naval Research Young Investigator.

Mailing address: 500 West 120th street, Columbia University, New York, NY 10027. Email address: `djd@cs.columbia.edu`.

X Through the Firewall, and Other Application Relays

G. Winfield Treese

MIT Laboratory for Computer Science and Digital Equipment Corporation

Alec Wolman

University of Washington and Digital Equipment Corporation

Abstract

Organizations often impose an administrative security policy when they connect to other organizations on a public network such as the Internet. Many applications have their own notions of security, or they simply rely on the security of the underlying protocols. Using the X Window System as a case study, we describe some techniques for building application-specific “relays” that allow the use of applications across organizational boundaries. In particular, we focus on analyzing administrative and application-specific security policies to construct solutions that satisfy the security requirements while providing the necessary functions of the applications.

1 Introduction

This paper presents some general techniques for making network applications available through secure gateways, based on experience from managing an Internet gateway at Digital Equipment Corporation’s Cambridge Research Laboratory (CRL). Like many organizations, Digital operates firewalls at its Internet gateways to isolate the internal network from the rest of the Internet. This isolation is only partial, because communication is the goal of the connection in the first place. Unfortunately, the Internet is not always a safe place, and some form of protection is necessary.

A firewall (sometimes called a “gateway”), such as those described by Cheswick [2], Ranum [11], or Schauer and Wolfhugel [13], is a collection of computers intended to protect an organization connected to a public network. The fundamental premise in the design of firewalls is that it is easier to secure a small number of systems rather than hundreds or thousands. A firewall isolates insecure systems inside the organization from the public network. We discuss the design of traditional firewalls in more detail below.

At CRL, we began with a traditional firewall system. CRL operates one of Digital’s connections to the Internet, and the firewall has been in place since that connection was established. At the outset, each person who needed access to the Internet also needed an account on a trusted firewall system. The only exceptions to this rule were those users who only needed to use electronic mail and USENET news, which were forwarded appropriately by a trusted gateway operated as part of the firewall system. Assigning user accounts does not scale very well, however; the administrative costs alone can be prohibitive. Because of this, only a small number of people were allowed to have accounts.

In order to provide some limited Internet access to more people, we began implementing “relays” for various applications, such as FTP, Telnet, and X. A “relay” is an intermediary program, specific to a given application, that permits users on the inside of the firewall to use services available on the public network. The relays run on trusted firewall systems.

The design of application relays is not merely a technical problem. An organization may have an “administrative security policy” that makes some statements about what is allowed and what is not. Relays must respect these policies. The challenge is designing a relay that both provides the needed application functions and satisfies the requirements of the security policy.

In this paper, we discuss some general principles for attacking the problem of designing relays to satisfy both technical and policy requirements. We use our experience with constructing a relay for the X Window System [14]

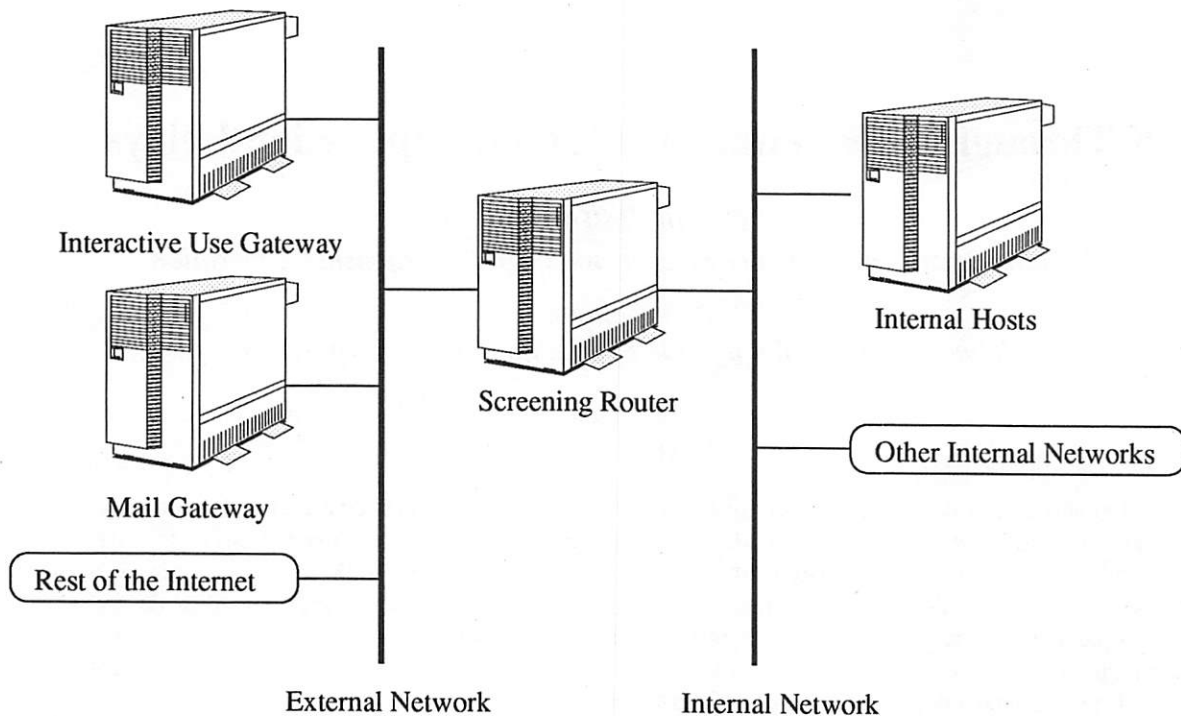


Figure 1: Typical Firewall Configuration

as a case study.

After some brief definitions, we begin with discussions of traditional firewall design and administrative security policies. Next we describe our approach to building the relay for X. This is followed by short descriptions of some other example applications. We compare this approach to some others from the research community and conclude with an evaluation of our experiences and some discussion of possible future work.

2 Definitions

In the sections that follow, an “internal” network is one inside an organization connected to the Internet, and an “external” network is the Internet. Note that “secure” as used here means some reasonable level of security, not an absolute high level. The goal of our work has been to extend services without sacrificing a significant portion of the security that already exists, whatever that might be for a given application.

We use the term “router” to refer to a device that operates on packets at the network layer. A “gateway” operates at the application layer (which may encompass the OSI [18] session, presentation, and application layers).

3 Traditional Firewall Design

A typical firewall configuration may consist of several computer systems. These include routers used to separate the internal and external networks as well as secure hosts that may be used for interactive use. A diagram of a typical firewall configuration is shown in Figure 1.

The routers used in the firewall are often capable of “screening” packets to select desired ones and discard undesired ones. Screening routers can be used to permit direct access between internal machines and external ones. One screening router implementation for UNIX systems is described by Mogul [7] (although not all of these have the flexibility of the one Mogul describes). Similar capabilities are now available in many commercial routers. Such systems allow a system manager to select packets based on some combination of protocol, source and destination IP address, and source and destination port numbers.

Some uses of screening routers include:

- Protecting firewall systems. Even though a system is part of the firewall, we may wish to prevent it from exchanging certain kinds of packets with external machines. For example, a dedicated mail relay may not need to allow telnet access from external machines. Putting it behind the screening router adds an extra layer of protection.
- Limiting interaction between firewall systems and internal systems. A firewall system probably does not need full access to all internal systems; its access should be limited to the functions that are required. Such a restriction limits the possible actions of an intruder who compromises the firewall system.
- Allowing widespread access for certain services. There may be a few services that are deemed sufficiently safe for all internal systems to use. Such services may include, for example, the FTP and WHOIS services operated by the Network Information Center (NIC). If we believe that the NIC is unlikely to be compromised, then there is little risk posed by allowing such access.

Screening routers can provide substantial protection, but there are many applications that are difficult to support using only screening routers. To circumvent this problem, firewall configurations often include a few privileged hosts that have broader access to the Internet (although their access may not be complete). Rather than allowing direct communication between internal and external machines for applications such as electronic mail and USENET news, the privileged machines accept mail or news messages and deliver them appropriately on the other side of the firewall. An application such as remote login requires a different solution — rather than allowing remote logins between all internal and external machines, a typical configuration would require all users to login to the privileged machine as an intermediate step. These user accounts add some administrative overhead to managing the gateway. This overhead is an important factor in how large the gateway can grow.

Firewall configurations of this nature are currently available from several vendors and consulting services.

4 Administrative Policies

Most organizations connected to the Internet have some sort of policy concerned with the security aspects of the connection. In many cases, common to universities, the policy is that no security should be assumed. Other organizations, such as companies or government agencies, frequently have detailed policies that limit use of the network. We will refer to such policies as “administrative policies.” Some examples of issues covered by administrative policies include:

- Who can use the connection. Typically only employees or those individuals otherwise associated with the organization may use the connection. Many organizations further limit the set to those who have authorized user accounts on certain computers associated with the connection.
- What kind of data flow can occur. Many organizations are concerned about potential flow of private information outside the organization (for example, the source code to a critical product). Others are concerned with the flow of data into the organization (for example, employees retrieving proprietary data from another company).
- Which applications can be used. Policies frequently limit the set of applications that can be used with a connection. These limits may be based on lack of a demonstrated need for the application, a belief that the application is unsafe (whether because of known problems, a guess, or a history of security problems), or expected violations of other aspects of security policy. For example, an outgoing file transfer utility may violate the policies surrounding the kinds of data flow that are permitted.

To the technically-oriented reader, such policies (or variations on them) may seem silly, excessive, or ineffective. In some cases, they are. In other cases, the issues may be more subtle. For example, information can flow out of a company in many ways, especially in the hands of a disgruntled employee, so restricting outbound data transfer may not seem reasonable. On the other hand, consider a thief who breaks into a corporate computers system using a 1200 baud modem. He cannot download a large quantity of source code quickly over such a connection, but he might be able to take advantage of a high-speed network connection to move the code out to some other machine on the network. In this case, restrictions on data transfer serves to limit the damage the thief can cause.

In any case, we are not concerned here with defining or defending particular choices of administrative policies. Given a policy, we want to analyze it in the contexts of particular applications in order to construct an appropriate gateway. These policies are not stated in a formal fashion; part of the problem is constructing a real system that satisfies a vague policy statement (and its issuers).

4.1 Example Administrative Policy

For our case study, we consider the following policy:

- Access is limited to those with authorized user accounts.
- Unauthorized individuals must not be allowed to gain access to internal machines or information.
- It must not be possible for individuals to transfer large amounts of information out of the organization at high speed. Individuals with authorized user accounts are trusted not to abuse their privilege to send out proprietary information.
- The gateway machines should keep reasonable logs about their use.

Like most administrative policies, these policies are general statements and do not prescribe specific details for implementation. Often it is important to understand the intent of the policy as well as its statement in order to construct a system that satisfies the makers of the policy. The policy described above, for example, may be understood to permit electronic mail, even though mail allows some outbound transfer of information. Mail is usually not high bandwidth, and some information about each message is usually logged.

In Section 7 we describe some other applications and variations of the administrative security policy.

4.2 Scope of Protection

In this paper we are primarily concerned with protecting applications. We do not address issues such as spying on individual packets on the Internet, the abuse of protocols to create covert channels for signaling, forging source or destination IP addresses, or "hijacking" a TCP connection by inserting packets with proper sequence numbers into the connection. Many of these potential problems with TCP/IP are discussed by Bellovin [1]. Although we do not consider them here, these potential problems should be considered when designing a real firewall system.

5 A Relay for the X Window System

Some applications are difficult to operate securely with traditional firewall configurations. In our case, the critical ones were applications using the X Window System. After implementing a solution to the X problem, we began to investigate other applications where we could apply some of the same general techniques.

In the X Window System, the basic security model allows a user to control the set of hosts allowed to make connections to the X server. This control only affects new connections, not existing ones. Many users disable the access control entirely for personal convenience when using a more than a few hosts.

Researchers at CRL often collaborate with researchers at universities, and they want to run applications on the university machines with the X display on a workstation at CRL. Because of X's weak security model, simply allowing all X traffic to pass through the screening routers would not meet our security requirements.

Since we had no technical way of enforcing the use of access controls, we had to assume that the internal users would not use the existing access control mechanism carefully. There are also hooks in the X protocol for passing other authentication data to the server; we explain below why that mechanism is not sufficient.

One approach might be to modify the screening routers as necessary to allow connections between specific machines. The main problem with this approach is the management overhead. To keep the screen up to date would require a system manager to take action whenever a new connection was needed or when an old one was no longer necessary. A privileged application to do this would require substantial access controls and authorization for the users.

For quite some time, if a researcher asked for this capability, the answer was that it was simply unavailable. Fortunately, researchers don't like to take "no" for an answer, especially when it has to do with getting their work

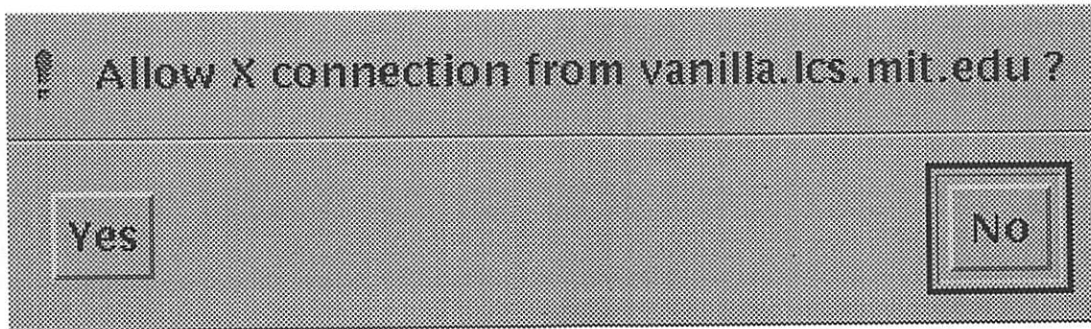


Figure 2: Dialog box for *xforward*

done, so we began thinking about how to provide this capability. We realized that we needed to solve the following problems:

- Limiting which systems could connect to an internal display. Since the address of the client host is the best information we have about the client, we can use it to restrict which hosts can connect.
- Ensuring that there were no unauthorized connections. Limiting connections to a small set of systems does not eliminate all problems. A hostile user on an allowed system might try to connect to the display. The X protocol allows a client to spy on the entire state of the screen as well as the input stream, so a client can silently eavesdrop on other applications.
- Not modifying any client or X server software. Since we have no control over either the clients or the servers, we could not require any modifications to the software. This ruled out modifying clients and servers to use a strong authentication system such as Kerberos [16] or SPX [17].

The first problem means that we are limited at best to a solution based on IP addresses, not authenticated individuals. That immediately led us to the second problem, of ensuring that no unauthorized connections were made by other users of the remote machines.

Our solution is a modified TCP protocol forwarder called *xforward*, which is run by a user on a privileged gateway machine. A TCP forwarder is a program that listens for data on a port and forwards anything it receives to another system. The screening router is configured to only allow X traffic between the internal hosts and the privileged gateway. *xforward* allows the user to specify a list of hosts that are allowed to connect to the X server. It establishes a relay between a pseudo-display number on the gateway machine and the user's workstation.

Foreign X clients attempt to establish connections to the pseudo-display on the gateway machine, where the list of allowed hosts is checked. When that check is successful, the connection must then be explicitly approved by the user through a dialog box created by *xforward*. The dialog box is shown in Figure 2. This makes the establishment of each connection an intrusive action and ensures that the user is aware of all connections created to the display from the outside, so an intruder cannot spy on the X server. In addition, the list of allowed hosts is specified at relay startup time, and there is no way to disable this mechanism or to specify wildcards. If there is no activity on any connection for a certain period of time, all connections are closed and the relay terminates. For reference, the *xforward* manual page is shown in Appendix A.

This solution may seem rather simple. Indeed, it is not a matter of deep thought. The interesting principle here, however, is that we can use specific knowledge about an application to create a relay that provides the required functions of the application while preserving a desired security policy. In the next section, we describe several examples in which the application semantics differ enough that *xforward* itself would not work, but the general principle of analysis applies.

6 Example Applications

Other applications that we have used in this way include FTP, Telnet, WHOIS, and *sup* (a software distribution program developed by the Mach project at Carnegie Mellon). In each case, we used a similar process of analyzing

the application's security policies and the administrative policies to develop a relay that provided the necessary functions while meeting the administrative policies. For comparison, the section concludes with analyses of electronic mail and USENET news in the same framework.

6.1 FTP

The File Transfer Protocol (FTP) [10] is widely used on the Internet for copying files. We may wish to allow many people on internal systems to copy files to and from external systems. With the policies we are currently considering, that is not possible without giving everyone an authorized user account on gateway machines. Suppose, however, that we relax the administrative policy slightly:

- Any employee can copy files from systems on the Internet to internal machines.

This rule does not differ much from the general intent of our original example. It does, however, give us some additional freedom to implement a gateway mechanism. There are two challenges here: one a technical problem and one caused by the administrative policy. The technical challenge is that FTP uses separate control and data connections, so a simple TCP relay is insufficient. The administrative challenge is that this policy does not permit copying files from internal machines to external machines, so even a simple FTP relay that can manage the data and control connections is not sufficient.

The solution in this case is an FTP relay that checks each FTP command before passing it through. Requests to store files are denied, and other requests are passed through. In addition, of course, the FTP relay limits access to internal machines. Such a relay has been implemented; a sample interaction with it is shown in Figure 3.

Given an FTP relay, however, we may be able to expand the offered service with an additional relaxation of the policy:

- An employee may transfer files out if the identity of the employee can be determined and recorded.

Authorized user accounts satisfy this requirement. We can also satisfy the requirement by using a strong authentication system, such as Kerberos [16], SPX [17], or "smart cards." In fact, a system has been implemented based on the the SecureNet SNK-004 from Digital Pathways, Inc. An individual with a registered key is allowed to transfer files after authentication. Figure 4 gives a sample dialog with this service.

Hence, with an understanding of both the administrative policy requirements and the details of the FTP implementation, we can devise a functional relay. If we understood the policy and had only a vague understanding of how FTP works, we might conclude that such a relay is not possible. If, on the other hand, we understood the FTP implementation and did not fully understand the policy, we might conclude that such a relay is not permissible.

6.2 Telnet

Telnet [9] is widely used on the Internet for remote logins. General telnet access between internal and external machines can serve many purposes, supporting cooperative work projects, customer support, and traveling employees who need to read their electronic mail. The relaxed version of the policy described for FTP is sufficient to permit a telnet relay implementation using a simple TCP relay and dialog to set up the connection, given some kind of strong authentication mechanism.

With "outbound" telnet (from an internal machine to an external one) we can take an extra precaution, however. In the abstract, we are supporting interactive remote terminal service, not arbitrary data connections. Hence, it may be reasonable to limit the data rate on the outbound connection to (say) 9600 bits per second. This is surely fast enough to handle a person typing; we are not interested in anything more.

6.3 WHOIS

WHOIS [6] is a TCP-based query/response service, often used as a directory service. The Internet Network Information Center (NIC) has historically maintained a name lookup service for individuals on the Internet (although the Internet is now so large as to make maintaining a centralized directory impossible).


```

% ftp ftp-gw
Connected to ftp-gw.
220-FTP passthrough server
220-To connect to a remote FTP server give your login as:
220-user@server.serverdomain (EG: anonymous@host.cs.mumble.edu)
220-
220-NOTE: All file transfers are logged by the relay, and are
220-      most likely also logged by the system at the other end
220      of the connection.
Name (ftp-gw:): anonymous@gatekeeper.dec.com
331 Guest login ok, send ident as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
...
README.nfs
...
226 Transfer complete.
448 bytes received in 0.18 seconds (2.4 Kbytes/s)
ftp> get README.nfs
200 PORT command successful.
150 Opening ASCII mode data connection for README.nfs (2799 bytes).
226 Transfer complete.
local: README.nfs remote: README.nfs
2853 bytes received in 0.54 seconds (5.2 Kbytes/s)
ftp> put README.nfs
200 PORT command successful.
530 Operation denied by FTP gateway
ftp> quit
221 Goodbye.

```

Figure 3: Sample interaction with FTP relay.

```

% ftp ftp-gw
Connected to ftp-gw
220-FTP passthrough server
220-To connect to a remote FTP server give your login as:
220-user@server.serverdomain (EG: anonymous@host.cs.mumble.edu)
220-
220-NOTE: All file transfers are logged by the relay, and are
220-      most likely also logged by the system at the other end
220      of the connection.
Name (ftp-gw): treese@somewhere.edu
331 Guest login ok, send ident as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> ftp> put fubar
200 PORT command successful.
530 Operation denied by FTP gateway
ftp> quote authorize treese
331 Enter response code for 88146:
ftp> quote response 35233348
230 Accepted and authorized, 305336
ftp> put fubar
200 PORT command successful.
150 Opening data connection for fubar (192.58.206.2,3441).
226 Transfer complete.
local: fubar remote: fubar
20929 bytes sent in 0.035 seconds (5.8e+02 Kbytes/s)
ftp> quit
221 Goodbye.

```

Figure 4: Sample interaction with FTP relay using authentication.

Given the somewhat relaxed policy described for FTP and telnet above, we consider access to the NIC WHOIS server. For this application, access to the NIC is sufficient because it holds the directory of interest. Widespread access to WHOIS servers throughout the network is a different issue. In this case, the protocol itself is not a means of data transfer, and we wish to provide the service for all internal users.

A simple TCP relay that only connects to the NIC suffices for WHOIS. It should limit access to internal systems, but no access control beyond that is necessary. It does not transfer significant data, and an intruder would need to compromise both the NIC and an internal machine to hijack the relay.

6.4 Sup

sup [15] is a software distribution program developed by the Mach project at Carnegie Mellon University. It is used to distribute updates to the Mach software to interested parties who are cooperating with CMU. Suppose we are interested in receiving updates to Mach on an internal system. *sup* is clearly designed for data transfer, so we cannot provide widespread and unauthenticated access. Since only one internal group needs to use *sup*, we can use a TCP relay that admits connections only from their machine and connects only to a designated machine at CMU. An intruder must compromise both ends of the connection to move data through the relay.

6.5 Electronic Mail and USENET News

Relays are nothing new to electronic mail and USENET news; “store-and-forward” systems have been relaying mail through organizational boundaries for years. We believe that these applications fit well into our framework. The security policy is relaxed to allow use by anyone for inbound or outbound messages. Messages “touch down” on a gateway machine, where sender and destination can be logged. The applications are not high in bandwidth because of the usual processing overhead.

7 Other Approaches

Application relays are not the only possible means to secure connected but distrustful networks. In this section we discuss some alternatives that operate at the network layer, concluding with a comparison to our work at the application layer. The approaches described here all interact with the routing of a packet through the network.

Routing determines the path a packet will take through a network. The path may traverse several networks and routers. Each router decides what action to take for each packet. This action may be to forward it to the ultimate destination, forward it to another router, or reject the packet for some reason (possibly with a notification to the sender). Most routers today try to select a route that minimizes some metric of the path, such as delay or number of routers traversed. Routers could use other considerations in making these decisions. For example, the router might permit only certain senders, or it may forward packets of certain users over higher-speed links. The alternative approaches discussed here vary primarily in what information is used to make routing decisions.

7.1 Screening Routers

As we have discussed, a screening router can be an important component of a firewall system. A screening router permits an organization to implement a variety of security policies [8]. Unfortunately, a screening router alone cannot always satisfy the administrative policy for a given application. Because it has no real knowledge of the application, it cannot perform application-specific actions such as those we use for *xforward*.

7.2 Policy Routing

Policy routing, such as the architecture described by Clark [3], enables routers to make decisions based on resource policies, including security as well as other considerations such as link speed or cost. Clark proposes an architecture in which policy routes are synthesized by the source host and its administrative domain. The routers along the path are responsible for enforcing the selected policy.

In practice, the main problem is that it requires wide deployment of both end systems and routers capable of creating and enforcing policy routes. Therefore we could not build a system based on policy routing.

7.3 Visa Protocols

Visa protocols [5] are another approach to managing the flow of packets between organizations. A “visa” is an unforgeable cryptographic stamp attached to a packet. In the same way that a passport visa grants permission to visit a country, a packet’s visa grants it permission to enter or leave an organization. Routers at organizational boundaries can check the visa of a packet before forwarding it to verify that it has the appropriate permissions. Visas are issued by trusted access control servers in the source and destination organizations.

Visa protocols solve many of the problems we have considered so far. However, the routers still cannot take action based on the application, which is required for some of the policies we have considered. For example, a router would not be able to create the *xforward* prompt for a new connection, since it has no knowledge of the protocols it is routing.

7.4 Why the Application Layer?

Estrin and Tsudik [4] argue that the network layer is the appropriate place for inter-organizational access controls. This conclusion is based on the design principle of the end-to-end argument [12]. The basic end-to-end argument is that controls should be placed at the highest layer of the network at the actual endpoints of the communication, since only the highest layer will actually be able to ensure properties such as reliability and security. Estrin and Tsudik note that one may consider entire organizations as endpoints in their own right, because the network resources of the organization must be protected as well as the end systems within the organization.

Because the network layer is the highest normally handled by an organization’s border routers, the end-to-end argument places responsibility for security at that layer. We have gone a step beyond, however, using application relays on border gateways. Given the application relays, the end-to-end argument moves the responsibility for security to the relays.

Application relays give us the additional flexibility to perform application-specific functions — *xforward* can interactively request approval for a connection, for example.

8 Conclusions and Future Work

Building secure and useful inter-organizational networks is a complex challenge. By building application-specific relays to satisfy particular administrative policies, we have been able to expand the services available to users lacking full access to the Internet. These services preserve the desired security of the internal network as well. Not every application is suitable for this kind of relay, but the techniques work in a surprising variety of situations.

In addition to the security issues, these techniques can be useful when routing issues would normally prevent access between networks. For example, if an organization uses a single network number for its internal network, it is limited to a single primary route from the Internet into the organization’s network, even if it has several points of connection. The application relays hide this problem from the end systems.

As the range of services available on the Internet grows, we hope to be able to make many new services available on protected networks by applying these techniques. Although the implementation often varies from application to application, making it difficult to reuse source code, each new application is easier to construct because it builds upon the cumulative experience. Much work remains to be done, however, in making this process more straightforward and automatic.

Although the telnet and ftp relays do eliminate the need for user accounts on the gateway, the current version of *xforward* does not. In order to eliminate the need for user accounts, we need a method for users to invoke the X relay remotely, without having a user account on the firewall. We might accomplish this in two ways: first, we can build a listener that will accept *xforward* invocation requests from internal machines, on the theory that anyone on an internal machine should have authorization to use the X relay. If this does not meet the administrative security requirements, in addition we can make use of “smart cards”, to verify that the requester is in fact authorized to remotely invoke the X relay.

All of the relays we have implemented use TCP. For TCP applications, the relevant security decisions can usually be made at connection creation time. Application-specific relays can also be built for applications that use UDP. For UDP applications, the relay’s decision whether or not to forward would have to be made packet by

packet, rather than per connection. This would require detailed information about the application specific protocol layered on UDP. This is similar in spirit to what the FTP relay does when it checks each FTP command before passing it through.

On a more practical note, we are planning to extend *xforward* to support compression of the X protocol for use over low-bandwidth network connections. We also plan to add an interactive control panel that will display status information and allow users to modify the access lists during execution.

9 Acknowledgements

The authors would like to thank John Kohl for implementing the original TCP forwarding code now used in *xforward*; it had no particular security features. Brian Reid forced us to carefully work through the security issues surrounding *xforward*. Marcus Ranum designed and implemented the FTP and Telnet relays and has provided much helpful discussion on these issues. Victor Vyssotsky gave us the freedom to experiment carefully with the Internet gateway at Digital's Cambridge Research Lab. Neil Fishman and Ted Wojcik assisted with our experiments at CRL's gateway. We would like to thank Murray Mazer, Larry Stewart, Jim Miller, and the USENIX referees for helpful comments on this paper.

References

- [1] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 9(2):32–48, April 1989.
- [2] Bill Cheswick. The design of a secure internet gateway. In *Proceedings of the USENIX Summer Conference*, 1990.
- [3] David D. Clark. Policy routing in internetworks. *Internetworking: Research and Experience*, 1:35–52, 1990.
- [4] D. Estrin and G. Tsudik. An end-to-end argument for network layer, inter-domain access controls. *Internetworking: Research and Experience*, pages 71–86, June 1991.
- [5] Deborah Estrin, Jeffrey C. Mogul, and Gene Tsudik. Visa protocols for controlling inter-organization datagram flow. *IEEE Journal on Selected Areas in Communication*, 1989.
- [6] K. Harrenstien, M. K. Stahl, and E. J. Feinler. NICNAME/WHOIS. RFC 954, Network Information Center, October 1985.
- [7] Jeffrey C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the USENIX Summer Conference*, pages 203–221, Baltimore, MD, June 1989.
- [8] Jeffrey C. Mogul. Using *screend* to implement IP/TCP security policies. Technical Note TN-2, Digital Equipment Corporation Network Systems Lab, 1991.
- [9] J. B. Postel and J. K. Reynolds. Telnet protocol specification. RFC 854, Network Information Center, 1983.
- [10] J. B. Postel and J. K. Reynolds. File transfer protocol. RFC 959, Network Information Center, 1985.
- [11] Marcus J. Ranum. A network firewall. In *Proceedings of the World Conference on System Administration and Security*, Washington, D.C., July 1992.
- [12] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2:195–206, 1984.
- [13] Hervé Schauer and Christophe Wolfhugel. An Internet gatekeeper. In *UNIX Security Symposium III Proceedings*, pages 49–61. USENIX, 1992.
- [14] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, Bedford, MA, 3rd edition, 1991.
- [15] Steven Shafer and Mary Thompson. The SUP software upgrade protocol. Available by anonymous FTP from mach.cs.cmu.edu:/pub/sup/sup.doc.

- [16] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication system for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, January 1988.
- [17] J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 232–244, Oakland, CA, May 1991.
- [18] Hubert Zimmermann. OSI reference model — the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, Com-28(4):425–432, April 1980.

G. Winfield Treese is a member of Digital Equipment Corporation's Cambridge Research Laboratory and a graduate student at the MIT Laboratory for Computer Science. Prior to joining Digital he worked at MIT Project Athena. He is currently working on high-performance networks. Win holds an S.B. in Mathematics from MIT and an S.M. in Computer Science from Harvard University. His electronic mail addresses are treese@crl.dec.com and treese@lcs.mit.edu. His postal address is Digital Equipment Corporation, One Kendall Square, Building 650, Cambridge, MA, USA 02139.

Alec Wolman is a graduate student at the University of Washington, currently on leave from Digital Equipment Corporation's Cambridge Research Lab. He holds an A.B. in Computer Science from Harvard University. His electronic mail addresses are wolman@cs.washington.edu and wolman@crl.dec.com. His postal address is: Computer Science and Engineering, University of Washington, Sieg Hall, FR-35, Seattle, WA, USA 98195.

xforward (local)

UNIX Programmer's Manual

xforward (local)

NAME

xforward – provide user-level X forwarding service

SYNOPSIS

xforward [options]

OVERVIEW

xforward provides a user-level X11 forwarding service which can be useful if there are IP network topologies which provide non-transitive routing (e.g. routers which implement policy packet screening).

OPTIONS

-display *display*

Specifies the destination display where the user wants applications to appear. Without this argument, *xforward* will use the DISPLAY environment variable.

-allow *allowed-host1* [*allowed-host2* ... *allowed-host16*]

Only connections from *allowed-hosts* are permitted. At least one *allowed-host* must be specified, and at most sixteen are allowed.

DESCRIPTION

xforward will choose an unused port for the local display, and listen for connections on the local host at that port. *xforward* informs the user which port to use as the local display, when *xforward* is first invoked. When it receives a connection, it will create a confirmation pop-up on the destination. If the user confirms the connection request, it will create a separate socket and connect it to the destination, and then commence data piping between the two connections. *xforward* can handle multiple simultaneous connections.

If there is no activity on a connection for 90 minutes, the connection is closed. If the X server at the destination does not have access control enabled, then *xforward* will report an error.

If a connection is closed by either end, any buffered data is drained to its destination before *xforward* will close the corresponding socket on the other end.

SEE ALSO

accept(2), bind(2), connect(2), listen(2), select(2), socket(2)

BUGS

Out-of-band data is ignored/thrown away.

If the initial connection to the destination fails for some reason, the client who connected to the local display will get an open and immediately closed TCP connection, which may cause some difficulty for programs that expect some sort of server response or an error code indicating failure to connect.

AUTHORS

John Kohl, MIT Project Athena and Digital Equipment Corporation
Modifications for X by Win Treese and Alec Wolman

The Ferret Document Browser

Howard P. Katseff

Thomas B. London

AT&T Bell Laboratories

Holmdel, NJ 07733

Abstract

The Ferret Document Browser is a vehicle for exploring the design and use of document storage and retrieval systems. Its distributed, modular structure allows independent information providers to control their data, yet make use of a common access and billing control facility. Document images are distributed via a nationwide AT&T corporate internet which consists mainly of Ethernet networks interconnected by leased data circuits. The relatively low bandwidth of this networks is dealt with by compressing the documents for transmission, and by decompressing pages as requested on the workstation. A page image can be decompressed and displayed in less than a half second.

A broadband version of the system makes use of the BBFS broadband file server, the HPC interconnect, the LuckyNet broadband network and the Liaison network multimedia workstation. This system allows document browsing at rates up to 15 page images per second.

1. Introduction

The Ferret Browser is a vehicle for exploring the design and use of network-based broadband, wideband, and narrowband image storage and retrieval systems. It provides wide area distribution of documents and images via a nationwide AT&T corporate IP network which consists of Ethernet and Frame Relay networks interconnected by gateways and leased data circuits. The relatively low bandwidth of this network is dealt with by compressing the images for transmission, and decompressing it as requested on the workstation. On a typical workstation, an image can be decompressed and displayed in less than a half second. The browser has been integrated with the AT&T Information Services Network's LINUS database system which provides document search and selection services and ensures that documents are viewed only as authorized.

Ferret's modular and distributed design permits the collaborative support of multiple information providers. In particular, the image databases are located at different AT&T locations and maintained by different organizations and the Ferret servers may be accessed from database systems other than LINUS.

The Ferret viewing software has been widely distributed to the AT&T R&D Community and is currently being used by nearly 1000 people. It runs on a wide variety of UNIX workstations using X windows, OpenLook, or Motif. It supports a wide range of industry standard document and image formats and currently provides access to several image databases, including 21,000 AT&T Bell Laboratories technical memoranda, and 9000 photographs from the AT&T archives. We also describe an experimental version of Ferret that gives far better performance: it can display pages at a rate of 15 per second via a broadband network.

2. Widely Accessible Browser

Since 1989, the AT&T Information Services Network has been scanning internal technical memoranda at 400 dpi (dots per inch) and storing the images on write-once optical disks^[1]. Approximately 21,000 documents are currently stored, and about 40 new documents are scanned each day. Requests for copies are fulfilled by printing the document on a 400 dpi printer at a central site and sending it out via company mail. While this system is a large improvement over its predecessor, where a clerk located the original document in a filing cabinet and

made a xerographic copy, there is still a several day delay before the requested copy is received. Requests for documents written before 1989 are still processed manually.

BBFS is a distributed broadband filesystem research effort^{[2] [3]} to support data-intensive applications, such as HDTV video. It is able to meet real-time constraints and stream data continuously at broadband rates. BBFS depends on distributed and parallel computing to provide the communications and processing needed to support resource-intensive applications. The current prototype system has 36 disks, providing more than 40 Gbytes of storage.

When the Ferret project was started, approximately 10,000 documents had already been scanned and stored on the Information Services Network's optical disks. This data was transferred to the BBFS file server via eight millimeter tape. Newly scanned documents are sent nightly via a local area network. The document images are stored as multi-page TIFF format files^[4] using CCITT Group 4 Facsimile-compatible compression^[5]. Without compression, our collection of 21,000 documents would require 1200 Gbytes, far exceeding the current capacity of the BBFS file server. With compression, only 30 Gbytes are needed.

In addition to storing images of documents at the original 400 dpi encoding, we also resample the images and store a 100 dpi version for display on a workstation. This is the lowest resolution readily visible on our workstations. Conveniently, the printed part of the document fits on the 1152x900 pixel screen of a Sun workstation at this resolution.

Photographs from the AT&T Archives are being scanned as time permits. Each is scanned in grayscale with a depth of 8 bits and a resolution of 62 dpi. The current TIFF software makes use of a Lempel-Ziv compression scheme that compresses a typical 8x10 photograph to 200,000 bytes. The images for the photos database are stored on a different Ferret server than the technical memoranda database. This Ferret server runs on a Sun workstation using ordinary magnetic SCSI disks. The image data currently requires 3 Gbytes of storage and is stored on 2 external disks.

AT&T employees have access to the LINUS (Library Network User Service) system. It provides access to many online databases, including the technical memorandum and photograph databases. For the technical memoranda, the citations in this database include authors, document numbers, keywords, and complete abstracts. Its Slimmer information retrieval system allows the database to be searched in a variety of ways. Once a document is chosen, the user issues a Slimmer command to view its image. In response, Slimmer determines which Ferret server provides documents for this database and sends it a message over the corporate network. A window for viewing the document is then created on the workstation. The file containing the images of the entire document, compressed at 100 dpi, is transmitted from the BBFS file server to a temporary file on the user's workstation. The transmission time depends on the length of the document and the speed of the communications line. Over the local Ethernet, a 25 page document requires less than 2 seconds to be transferred. When traversing a 64 kbit/sec interlocation link, the same document takes a minute to be sent.

To provide the illusion of instantaneous access, the browsing window is created while the file is being transferred. The window is usually displayed before the transfer of the compressed image is complete. While the document is being sent to the workstation, the Ferret browser allows the pages that have been already transferred to be viewed. Because the document is stored by ascending page number, the first page is usually transferred by the time the window is created. This allows the browser to display the first page of the document without delay. As long as the user does not immediately try to read a page at the end of the document, it gives the appearance that the file has been transferred in a few seconds.

As shown in Figure 1, Ferret has a simple user interface. The main feature is a large window (850x1100 pixels) which contains a page of the document encoded at 100 dpi. To its left is a slider bar that indicates the current page in the document, both by the height of the slider and by a small number just above the slider. The mouse buttons are used to traverse the document. The right button goes to the next page and the left button goes to the previous page. The middle button is used to move to an arbitrary page, indicated by the vertical position of the mouse pointer relative to the slider. If the right mouse button is clicked while the control key is held down, the document is continuously scanned in the forward direction. Similarly, the left mouse button is used to continuously scan backwards through the document. The window is destroyed by clicking the middle mouse button while the control key is held down.

The browsing software makes use of the compressed file that was copied to the workstation's local filesystem and decompresses each page as requested for display by the user. This technique is feasible because modern workstations can quickly decompress Group 4 Facsimile encoded files. For instance, a Sun IPX workstation takes less than a third of a second to decompress and display a single page. Further, the compressed file for an average document is small, about 15 kbytes to represent each page, or 300 kbytes for a 20 page document, so it is likely to reside in the file system cache so that no disk accesses are required.

3. Slow Ferret

We can also serve users who are not connected to the corporate network or do not have a terminal with windows or bitmap display capabilities, but do have access to a FAX machine. Documents are rescaled to 200 dpi for transmission via facsimile "fine" mode. The grayscale images of the photographs are rescaled to this size and dithered before transmission via a FAX modem, consuming several minutes of computer time. Most photographs look surprisingly good when printed on the FAX.

4. High Performance Browser

The high performance version of the Ferret document browser is designed to explore the feasibility of browsing through documents at high rates, up to 30 page images per second, providing the electronic analogue of flipping pages in a book. It makes use of the HPC local area multicomputer system^[6]. The current HPC/VORX configuration provides communications and distributed processing with 80 Motorola 68020 single board computers and 10 Sun hosts with a bandwidth of 113 Mbit/sec to each network node.

Long distance broadband communications is provided with the LuckyNet^[7] system. LuckyNet currently provides connections between three AT&T Bell Laboratories sites: Holmdel, Crawford Hill, and Murray Hill, with a total bandwidth of 452 Mbit/sec. The 5 km distance between Holmdel and Crawford Hill is spanned with a multi-fiber cable mounted on telephone poles and the 37 km link from Crawford Hill to Murray Hill is provided by line-of-sight super-high frequency (SHF) radio. The HPC switch is distributed among the three sites. Its VORX operating system provides seamless computing and communications among these locations.

Documents are displayed on the Liaison networked multimedia workstation^[8]. The prototype Liaison workstation is able to simultaneously display several windows with 30 frame per second video, each arriving from a different processor via the HPC. Its display is based on a Synergy Microsystems PEGC video board with a 1280×1024 pixel frame buffer connected to the local bus of its 33 MHz Motorola 68020-based single board computer.

The decompression software described previously is too slow to be used to decompress page images on the fly. Two feasible solutions to this problem are to use parallel processing to speed the decompression, and to decompress the document once and store its bitmap in a cache. Our first experiments have been with the latter approach.

Because the storage required by the images of a complete document may exceed the amount of local memory in our processors, we are forced to cache the document on disk. or speed, we make use of disk striping^[9], a technique that allows the parallel operation of several disks. When a document is chosen for viewing, the entire document is obtained from the BBFS file server and decompressed into the bitmap page images. The bitmap images are sent to a striped file on the BBFS file server. The file is striped across several disks round-robin, by scan line, to allow for parallel access when the file is read. When the user requests a page to be displayed, the disks send data simultaneously to the Liaison workstation via the HPC interconnect.

The current configuration uses four disks for parallel access and allows the document to be displayed at speeds up to 15 page images per second. The user interface is similar to that of the widely accessible browser. The major addition is a slider bar on the right side that allows the images to be paged forwards and backwards at various rates. Surprisingly, individual images can still be discerned at the rate of 15 pages per seconds.

5. Implementation Details

The widely accessible browser is implemented by several computers on the corporate internet, as shown in Figure 2. The LINUS system runs on a network of Sun workstations. It performs authentication and allows a user to search numerous databases interactively via Slimmer from home or office. It can be accessed via the *rlogin*^[10] command from a workstation on the corporate internet.

The Ferret server for the technical memoranda database runs on a Sun 4/370 workstation, *ferret*, that serves as a gateway between the internet and the HPC/LuckyNet interconnect. Ferret accepts TCP connections from workstations on the internet and services requests from the workstations to access the document images. Note that the broadband characteristics of BBFS are not needed for this server. We use BBFS because of its large storage capacity.

The Ferret browser must be run on a workstation running the X window system^[11] or one of its variants. The browser is started by running the *linus* program from the shell in a window. *Linus* opens the server end of a TCP connection that will ultimately connect to the LINUS system and starts two processes. One process waits for this TCP connection to be established and the other executes *rlogin* to connect to the LINUS system. After logging in to LINUS, the user can access many databases. Currently, the internal document database allows access to the bitmap images. The *view* command has been added to LINUS for a user to initiate the viewing of a document.

The *view* command opens a connection from LINUS to the waiting process on the workstation and sends a message with information on how to access the document. That process opens a TCP connection to *ferret* and uses the information obtained from LINUS to request the document image. It then copies the image (which is compressed in G4 fax format) from *ferret* to a temporary file on the workstation.

While the file is being copied, yet another process is created. This *browser* process creates a large window on the workstation and acts as the user interface for reading the document image. The *browser* process reads the temporary file and decompresses pages for display as they are requested. To allow it to run concurrently with the file transfer from *ferret*, the browser is able to defer the display of a page until it appears in the temporary file.

The implementation of the high-performance browser is similar. The major difference is that, instead of copying the compressed image over the TCP connection, *ferret* decompresses the file with the output directed to a four-way striped file on BBFS. While it is decompressing, the Ferret browser is started on the Liaison workstation. Figure 3 shows how the striped file is sent from BBFS to the workstation. Each square box in the diagram indicates a processor obtained from a pool of free processors. The processors labeled *diskfs* run a program that communicates with the X program controlling the user interface. Each of the *diskfs* processors respond to a request to display a new page by copying their portion of the image to the *vfilter* program. The *vfilter* is a standard part of the Liaison workstation and is responsible for the positioning, clipping, and synchronization of images destined for the workstation^{[8][12]}. In particular, it assures that the transmission of video to the workstation is synchronized with the monitor refresh.

6. Detail Mode

The Ferret browser normally displays monochrome images sampled at a resolution of 100 dpi. While text in these images is legible, the low resolution makes small fonts hard to read. Sampling the data at a finer resolution and directly displaying it on the screen results in an image much larger than the workstation screen, making it necessary to move the image left and right to read each line of text. This approach was rejected because it is cumbersome to use.

To provide a more detailed display, the browser makes use of the ability of most color and gray-scale workstations to display each pixel on the screen in varying intensities of gray. To present a detailed image of a document, the browser retrieves the 400 dpi image from the file server. It converts it to a 100 dpi image by mapping each four by four square of pixels in the 400 dpi image to a single pixel in the 100 dpi image. The intensity of the single pixel is made proportional to the number of white pixels in the 400 dpi image.

This technique produces an image more legible and aesthetically more pleasing than the normal 100 dpi monochrome image. The drawback is that it takes several seconds to process and display a high-detail page, as opposed to a half second for a monochrome page. More processing is needed because the 400 dpi image takes longer to transfer and decompress than the 100 dpi image and because Ferret needs to describe the image to the X window system using one byte per pixel instead of one bit per pixel.

Because of these significantly longer latencies, the browser normally displays images in monochrome. However, the detailed image of a page may be requested at any time. There is also a "detail mode" in which, whenever a new page is requested, it is first displayed in monochrome and then its detailed image is processed and displayed.

7. Printing

Because it is sometimes useful to have a printed copy of a document, Ferret includes a command that sends a copy of the document to a local printer. Most of the available printers make use of the PostScript language. However, the PostScript printers that we tried were far too slow to print 400 dpi full-page images, taking between 5 and 15 minutes to read the image of a single page from the Ethernet, process it and print it.

Our printing facility makes use of the Sun SPARCprinter, a low-cost laser printer that connects to the internal bus of a Sparc workstation. The printer does no image processing. Instead, it accepts bitmap page images from the workstation and prints them. The manufacturer's intent is to run page description software such as PostScript on the host and to ship its output to the printer. We circumvent this processing by decompressing our 400 dpi pages images and sending them directly to the printer. This permits us to print our documents at eleven pages per second, the rated speed of the printer.

8. PostScript documents

Many word processing systems produce output in the PostScript page description language^[13]. Many laser printers understand this language and can print documents in PostScript. PostScript document can be displayed on a workstation running the X window system with a PostScript viewing program.

For documents in PostScript, the Ferret system stores the PostScript version of the document. PostScript has some advantages. The PostScript version is shorter, requiring less disk space and less time to transmit to the user's workstation. Also, a single PostScript representation suffices for printing on devices with different resolutions and capabilities such as color.

Unfortunately, PostScript is not ubiquitous. Not all workstations have vendor-supplied PostScript viewing programs. A public domain viewer is available, but it is slow and has a limited selection of fonts. Further, PostScript output is not always portable. In addition to the font problem mentioned above, some documents do not display correctly with some viewing programs, presumably due to bugs either in the document or the viewer.

Because of these problems, when a PostScript document is entered into Ferret, the document is rendered into a compressed 100 dpi bitmap image that is stored on the file server in addition to the PostScript version. Each user can configure Ferret to specify a PostScript viewing program. If a viewer is specified, the PostScript document is sent to the workstation for display by the viewing program. Otherwise, the compressed bitmap image is sent to the workstation for display.

9. Conclusions

User reaction to the Ferret System has been positive. In April, 1992 a survey of 140 Ferret users was conducted^[14]. We initially thought that Ferret would be mainly used as a screening mechanism to find documents of interest. Instead, we find that over 95% of the survey respondents read the documents from their computer screens, instead of requesting printed documents. People remarked that it was faster and more efficient to read documents from their screens, and that they no longer felt a need to keep their own paper copies. We were also pleased to note that while most people accessed Ferret infrequently, on an average of once a week, most found the system to be easy to use.

Our design decision to provide speed instead of beauty was the correct one. The 1/3 second delay between clicking the mouse and seeing the next page is barely perceptible, making the browsing process more comfortable and natural. As evidenced by the user survey, the 100 dpi resolution seems to be adequate for reading documents from the display. However, as technology improves, this is one of the first things that we would upgrade.

Document browsing is coming of age. We know of other efforts to provide document images to users via data networks, including projects at the U. S. Patent Office and Carnegie-Mellon University. At AT&T, the Right-Pages system^[15] provides alerting and browsing for journal articles. The most significant difference between Ferret and these systems is that with Ferret, the information retrieval system is decoupled from the image viewing system. This makes it easy to provide a variety of front ends to access the same images. For instance, within AT&T, the technical community could make use of a sophisticated keyword based system, but managers, who may be less computer-literate and less familiar with the details of the subject matter, may have trouble using such a system. Instead, they could make use of a point-and-click graphical system which helps guide them to the information they need.

Another advantage of the decoupling is that different Ferret image databases may reside in different locations. We have observed that many information providers prefer to actually own the equipment that stores their images, and have it physically close by.

The Ferret Document Browser has become both an interesting research tool and a useful service. The high performance version allows us to experiment with the implications of presenting document pages at high rates and the widely accessible version allow the AT&T R&D community electronic access to a useful set of documents.

Future plans include allowing for the use of color images and higher resolution displays. Applications like catalog shopping and newspaper viewing will benefit from these enhancements. Faster networks, interfaces, and decompression software will be necessary to maintain reasonable performance.

10. Acknowledgements

We would like to thank Bob Gaglianello who suggested that the Ferret browser could be made widely available. The "back of the envelope" calculations performed in these discussions showed that the AT&T R&D Internet is fast enough to provide reasonable response time. Other people also provided invaluable help. Bill Austin was our primary liaison with the Information Services Network. Beth Robinson and Bruce Hillyer provided an early version of the BBFS file server for our use. Robert Waldstein integrated our software with the production LINUS system. Jan Wolitzky, Carlos Cruz, and Bill Boehm helped transfer the initial set of 10,000 documents to BBFS. Henry Shen helped port the software to an Intel 386-based workstation.

REFERENCES

1. Austin, W. E., and Wolitzky, J. I., "Electronic Document Management System for AT&T Proprietary Technical Information," *INFORM Magazine*, June, 1991.
2. Hillyer, B. K., and B. Robinson, "Aspects of the BBFS Broadband Filesystem," *Proc. First International Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1991.
3. Hillyer, B. K., and B. Robinson, "Communications Issues in BBFS, a Broadband Distributed Filesystem," *Proc. Globecom '91*, Phoenix, AZ, December 1991.
4. "Tag Image File Format Specification—Revision 5.0," Aldus Corporation, August 8, 1988.
5. Hunter, Roy, and A. Harry Robinson, "International Digital Facsimile Coding Standards," *Proc IEEE* 68,7, July, 1980, 854-867.
6. Gaglianello, R. D., et. al., "HPC/VORX: A Local Area Multicomputer System," *Proc Ninth Internat Conf on Distr Comput Sys*, June 1989, Newport Beach, 542-549.

7. Gitlin, R. D., et. al., "LuckyNet: An Overview," *Proc. Globecom '91*, December 1991, Phoenix, 1055-1064.
8. Katseff, H. P., et. al., "Experiences with the Liaison Network Multimedia Workstation," *Proc USENIX Symp on Experiences with Distr and Multiproc Syst*, Atlanta, March 1991, 341-350.
9. Kim, M. Y., "Parallel Operation of Magnetic Storage Devices: Synchronized Disk Interleaving," *Fourth Internat Wkshp on Datab Mach*, D. J. DeWitt and H. Boral, eds., Springer Verlag, New York, 1985, 300-330.
10. *UNIX Programmer's Manual*, 4.2 Berkeley System Distribution, Vol. 1, Computer Science Division, University of California, Berkeley, August 1983.
11. Scheifler, R. W., and Gettys, J., "The X Window System," *ACM Trans on Graphics* 5,2, April 1986, 79-109.
12. Katseff, H. P., and R. D. Gaglianella, "On the Synchronization and Display of Multiple Full-Motion Video Streams," *Proc IEEE TriComm '91: Communic for Distrib Applicat and Syst*, Chapel Hill, April 1991, 3-9.
13. "PostScript Reference Manual," Adobe Systems Inc., Addison-Wesley, 1985.
14. Austin, W. E., and Lunas, L., personal communication.
15. Story, G. A., et. al., "The RightPages Image-Based Electronic Library for Alerting and Browsing," *IEEE Computer*, Sept 1992, 17-16.

Biographical Information

Howard Katseff received his B.S. Degree in Computer Science at Cornell University in 1974. He did graduate work in theoretical computer science at the University of California, Berkeley and received his Ph.D. in 1978. Since then, he has worked at AT&T Bell Laboratories in Holmdel, New Jersey. He wrote the UNIX[®] debugger *sdb* and worked on the design and implementation of multiprocessor computer systems. He is now investigating applications for broadband networks. He may be reached at hpk@research.att.com.

Thomas London received his B.A. Degree in Mathematics from the University of Pennsylvania in 1972. He received his M.S. degree in 1974 and his Ph.D. degree in 1976 from Cornell University studying aspects of security and protection in computer systems. Working in Computer Science research at AT&T Bell Laboratories in Holmdel, NJ, since 1976, he has conducted research in operating systems, multiprocessor programming and systems, and communications intensive services and systems. He may be reached at tbl@research.att.com.

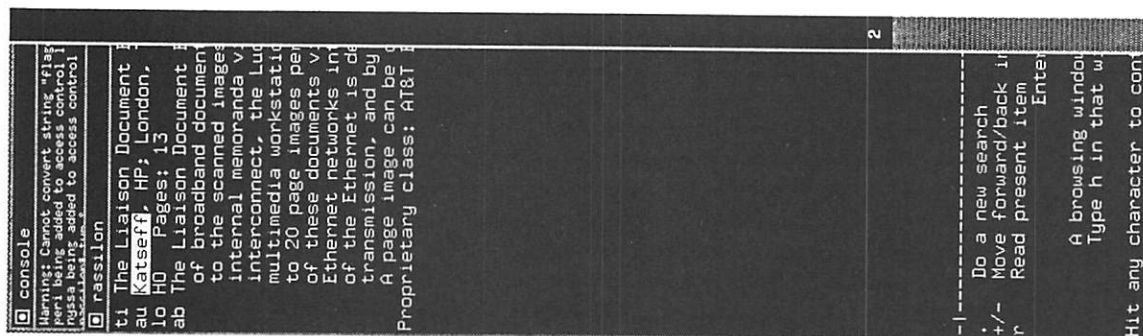


Figure 1. Typical display from the document browser.

Existing Infrastructure

Since 1989, the AT&T Library Network has been scanning internal technical memoranda at 400 dpi (dots per inch) and storing the images on write-once optical disks. Approximately 10,000 documents are currently stored. Requests for copies are fulfilled by printing the document on a 400 dpi printer at a central site and sending it out via company mail. While this system is a large improvement over its predecessor, where a clerk located the original document in a filing cabinet and made a xerographic copy, there is still a several day delay before the requested copy is received. Requests for documents written before 1989 are still processed manually.

AT&T employees have access to an online database of containing bibliographic citations for these memoranda. The Slimmer information retrieval system [4] allows the database to be searched in a variety of manners. Slimmer includes a command that arranges for a copy of the document to be printed and mailed to the user's office. There is a single database server for the AT&T R&D community that runs on a 75 MIPS Amdahl 5800/800E computer.

We have access to the HPC local area multicomputer system [5]. The current HPC/VORX configuration provides communications and distributed processing with 80 Motorola 68020 single board computers and 10 Sun hosts with a bandwidth of 113 Mbit/sec to each network node. Long distance broadband communications is provided with the LuckyNet [6] system. LuckyNet currently provides connections between three AT&T Bell Laboratories sites: Holmdel, Crawford Hill, and Murray Hill, with a total bandwidth of 452 Mbit/sec. The 5 km distance between Holmdel and Crawford Hill is spanned with a multi-fiber cable mounted on telephone poles and the 37 km link from Crawford Hill to Murray Hill is provided by line-of-sight super-high frequency (SHF) radio. The HPC switch is distributed among the three sites. Its VORX operating system provides seamless computing and communications among these locations.

BBFS is a distributed broadband filesystem research effort [7] [8] to support data-intensive applications. It is able to meet real-time constraints and stream data continuously at broadband rates. BBFS depends on distributed and parallel computing to provide the communications and processing needed to support resource-intensive applications. The current prototype system has 24 disks, providing more than 20 Gbytes of storage.

Liaison refers to both an architecture for a networked multimedia workstation and a prototype workstation that uses this architecture [9]. The workstation makes use of a distributed design that is driven by the need to allow many windows to be simultaneously displayed at video rates. The Liaison display performs only the low-level functions of acquiring and displaying images whereas the remainder of the workstation's functionality is performed by a pool of distributed processors. The prototype Liaison workstation is able to simultaneously display several windows with 30 frame per second video, each arriving from a different processor via the HPC. Its display is based on a Synergy Microsystems PEGC video board with a 1280x1024 pixel frame buffer connected to the local bus of its 33 MHz Motorola 68020-based single board computer.

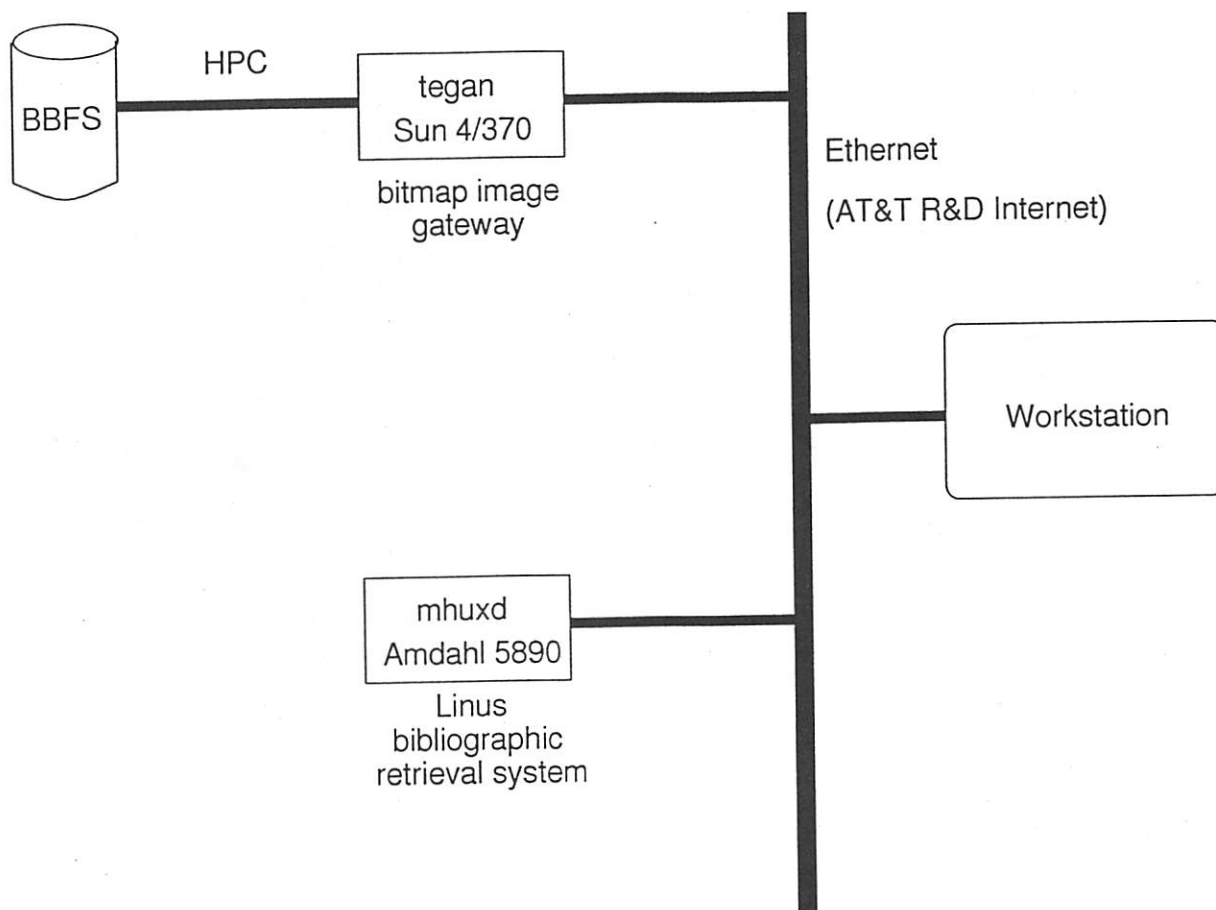


Figure 2. Architecture of the widely available version of the document browser.

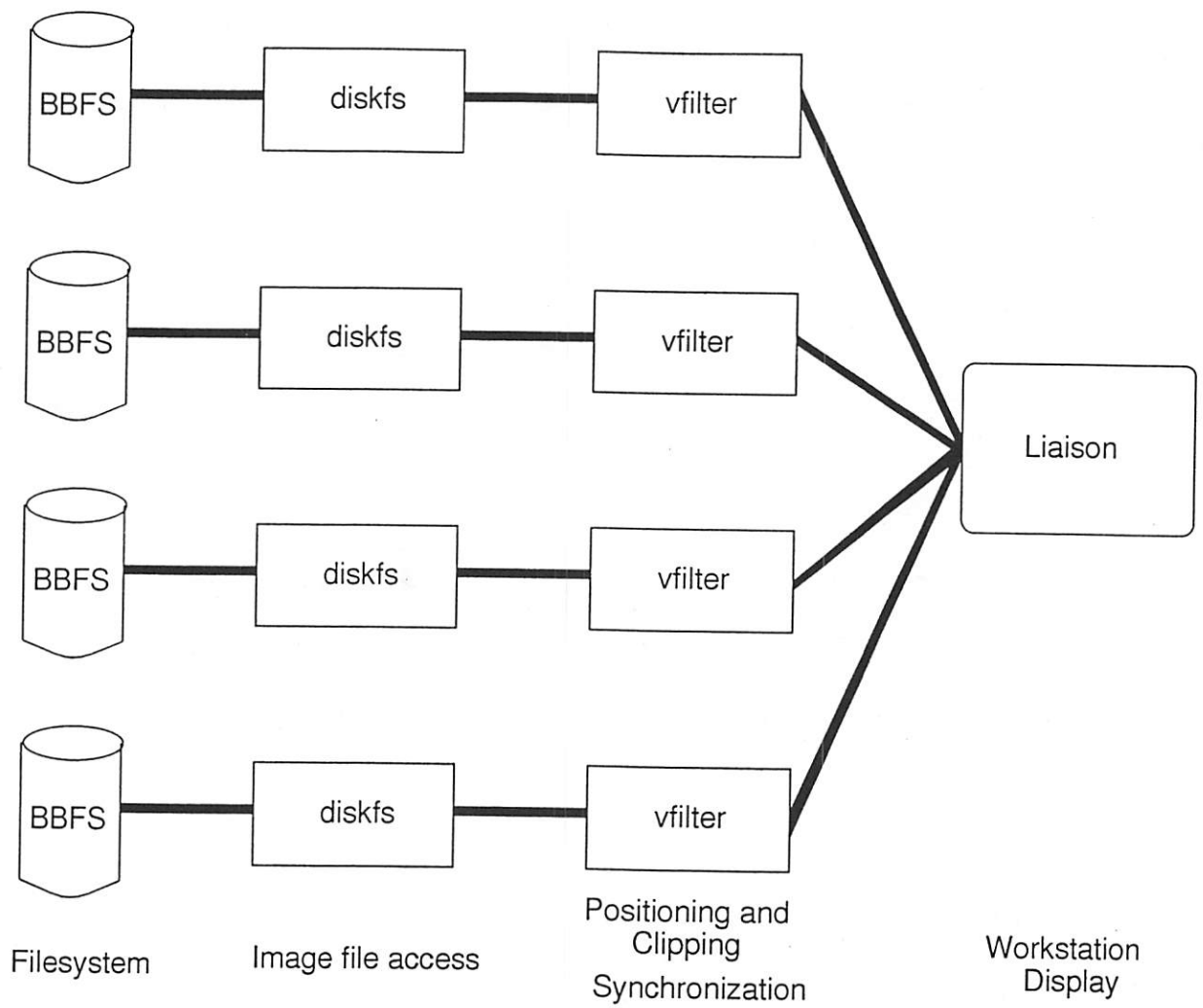


Figure 3. File access in the high performance version of the document browser.

LADDIS: The Next Generation In NFS File Server Benchmarking

Mark Wittle
Data General Corporation
Bruce E. Keith
Digital Equipment Corporation

Abstract

The ability to compare the performance of various NFS¹ file server configurations from several vendors is critically important to a computing facility when selecting an NFS file server. To date, nhfsstone² has been a popular means of characterizing NFS file server performance. However, several deficiencies have been found in nhfsstone. The LADDIS NFS file server benchmark has been developed to resolve nhfsstone's shortcomings and provide new functionality. The Standard Performance Evaluation Corporation (SPEC³) released the System File Server (SFS) Release 1.0 benchmark suite, which contains 097.LADDIS, as an industry-standard NFS file server benchmark in April 1993. This paper describes the major technical issues involved in developing the benchmark and the rationale used to establish default 097.LADDIS workload parameter values. Where appropriate, areas for further research are identified and encouraged.

1. LADDIS Overview

LADDIS is a synthetic benchmark used to measure the NFS [Sandberg85] request response time and throughput capacity of an NFS file server. The benchmark produces response time versus throughput measurements for various NFS load levels. LADDIS is executed concurrently on one or more NFS client systems, located on one or more network segments connected to the NFS file server being measured. The NFS client systems, called LADDIS load generators, send a controlled stream of NFS requests to the server, according to a specific operation mix and file access distribution, allowing precise measurement of the server's response time for each request. Other than basic server configuration requirements, LADDIS is unconcerned with *how* the file server provides NFS service - the benchmark measures the server as if it were a black box.

This paper describes the major technical issues involved in developing the benchmark and the rationale used to establish the default LADDIS workload parameter values, reflected in the 097.LADDIS benchmark released in the SPEC SFS Release 1.0 benchmark suite [SPEC93].

1.1. Nhfsstone Deficiencies

To date, nhfsstone has been a popular means of characterizing NFS file server performance [Legato89], [Shein89]. Nhfsstone synthetically duplicates average NFS file server resource utilization levels attained with a real user-level workload running on a collection of NFS clients [Keith90]. Despite its popularity, nhfsstone has several shortcomings.

Nhfsstone is sensitive to differences in NFS client kernel implementations across vendors' NFS clients. For a given logical file operation, nhfsstone can produce a different NFS protocol request sequence depending on the vendor's NFS client kernel and hardware configuration. As the NFS request load level produced by nhfsstone is increased, the impact of the client kernel implementation is amplified. This results in considerable inconsistency in the mix of operations generated at high NFS request load levels.

Nhfsstone attempts to overcome these difficulties through specific algorithms designed to by-pass NFS client kernel performance factors such as file name, attribute, and data caching. These work-arounds are operating-system specific and are not equally effective across different vendor platforms. The resulting variations can skew the performance results obtained with nhfsstone.

The use of operating-system specific algorithms and the need to access kernel data structures to monitor the delivery rate of requests to the NFS file server make nhfsstone difficult to port to a broad set of NFS client platforms.

Nhfsstone's load generating capacity is confined to a single NFS client system, which eliminates the performance impact of all network contention effects observed when servicing multiple NFS clients [Stern92]. Nhfsstone is limited in its ability to characterize NFS file servers that support multiple networks. An adequate solution to coordinating multiple nhfsstone instantiations requires the creation of additional control and results-consolidation software. The development of such control and results-consolidation software introduces second-order issues, primarily file set size control when using multiple clients to generate an aggregate load.

Finally, nhfsstone lacks a standardized approach to running the benchmark and reporting performance results that ensures fair comparisons of NFS file server performance across a range of configurations and vendor platforms.

1.2. LADDIS Improvements

To resolve nhfsstone's shortcomings, the LADDIS NFS file server benchmark was developed by a small group of engineers from NFS file server vendors: Legato Systems, Auspex Systems, Data General, Digital Equipment, Interphase, and Sun Microsystems. The benchmark was adopted and further refined by the Standard Performance Evaluation Corporation (SPEC) as an industry-standard NFS file server benchmark. SPEC released its System File Server benchmark suite, SFS Release 1.0, which comprises the LADDIS NFS file server benchmark, designated 097.LADDIS, in April 1993. All specific algorithms and parameter settings described herein apply to the 097.LADDIS benchmark.

The LADDIS NFS file server benchmark overcomes nhfsstone's shortcomings:

- NFS client kernel sensitivities were reduced significantly by implementing the NFS protocol within the LADDIS benchmark.
- NFS load generation algorithms were improved to produce a highly accurate mix of NFS operations.
- Control and result-consolidation functionality now supports coordinated, simultaneous use of multiple NFS client systems on multiple networks to generate an aggregate NFS load on the server under test.
- SPEC's SFS Release 1.0 Run and Report Rules for LADDIS provide a consistent and platform-independent method for running the benchmark and reporting NFS file server performance results, thereby eliminating confusion in interpreting NFS performance results within the NFS community.
- LADDIS has been ported to a variety of BSD-, SVR3-, SVR4-, and OSF/1⁴-based systems as part of SPEC's benchmark adoption process.

LADDIS also provides new capabilities. The NFS server file set targeted by LADDIS scales with requested NFS load level. This scaling allows LADDIS to mimic real NFS server environments where more server files are accessed as NFS load increases due to additional users or more NFS-intensive applications.

LADDIS provides new parameters which further refine nhfsstone's NFS workload abstraction of an NFS operation mix and an NFS operation request rate. NFS request packet sizes, data stream I/O lengths, file working set size, and file update patterns are under parameterized control. Users and developers can tailor the values of these parameters enabling LADDIS to produce a synthetic workload that emulates the load on NFS file servers in their own specific computing environment.

1.3. Dual-Purpose Tool

LADDIS is both an NFS server benchmark and an NFS performance characterization tool. As an industry-standard benchmark, SPEC SFS Release 1.0 Run and Report rules ensure that NFS server performance is measured consistently throughout the NFS community using standard, required workload abstractions that establish specific values for LADDIS's numerous workload parameters. Furthermore, SPEC SFS Release 1.0 Run and Report Rules ensure that NFS server performance results are reported in a thorough, consistent, results format, which specifies all of the hardware and software configuration parameters used to produce the results.

As a performance characterization tool, the breadth of LADDIS's workload parameters coupled with its multiple NFS client and network load generation capability allow NFS servers to be stressed under a variety of workloads. These capabilities facilitate the investigation of server behavior and performance in a number of computing environments.

The default LADDIS workload is based on an intensive software development environment. The parameters have been established based on the findings of several independent NFS file server studies. Future work is encouraged to develop additional LADDIS work load parameter sets that will allow LADDIS to generate synthetic workloads mimicing computing environments such as CAD, imaging, animation, and commercial NFS applications.

1.4. How LADDIS Measures NFS Server Performance

LADDIS measures the response time of the NFS server at the load generator using client operating system clock routines. Nearly all configuration and runtime parameters associated with the clients and networks are controlled explicitly by the benchmark. The goal is to isolate and measure the NFS server's response time by eliminating, minimizing, or controlling outside factors, including the client platform, whenever possible. With few exceptions, LADDIS is unconcerned with the file server's configuration parameters except to require that all relevant hardware and software details are reported with the performance results.

LADDIS implements the NFS protocol within the benchmark, rather than using the client kernel's NFS implementation. The benchmark formulates Open Network Computing (ONC¹) remote procedure call (RPC) packets directly, and measures server response time on entry and exit from the user-space RPC library calls used to send and receive network packets. This approach removes the client kernel's NFS attribute and data caching implementation from the benchmark's code path, yet maintains the RPC library as a standard base of software portability.

Some client operating system overhead is inherent in the process of generating NFS load and measuring response times. However, implementing the NFS protocol within the benchmark reduces operating system overhead, which has a positive impact on the accuracy of the benchmark. First, LADDIS gains precise control over the NFS request stream actually sent to the server. Second, much of the client kernel code path execution is removed from server response time measurements.

1.5. What LADDIS Does Not Measure

LADDIS is not an NFS client or NFS client-server benchmark. LADDIS uses NFS client systems to generate carefully controlled NFS load. Unlike normal client-side implementations of NFS, the NFS protocol is implemented in user space without attribute or data caching. The benchmark is meant to generate NFS load as it is seen by the NFS file server, not as it is indirectly generated by NFS client applications. This design choice makes LADDIS an inappropriate indicator of NFS client performance.

LADDIS is not an I/O bandwidth benchmark. The NFS protocol supports 17 different RPC request types. NFS READ and WRITE requests make up only about a third of the requests performed when using the benchmark's default workload parameters. Although LADDIS can be configured to perform a mix of operations dominated by READ and WRITE requests, other benchmarks tailored for sequential I/O bandwidth measurement are better suited to that task.

LADDIS does not model any specific user application. Executed with the default parameter settings, LADDIS generates a workload which represents an intensive software development environment, not any particular application or class of applications. LADDIS is also a powerful performance characterization tool that provides a wide range of workload parameters that can be adjusted to represent a specific environment. However, LADDIS is not a workload profiling tool. If users desire a non-standard workload, they must determine an appropriate LADDIS parameter set (workload abstraction) to represent their specific environment.

2. LADDIS Architecture

LADDIS benchmark components consist of a set of LADDIS load generators which create the NFS request load, a central LADDIS manager which coordinates execution of the benchmark, and the NFS file server under test. The LADDIS components execute on systems configured as NFS clients of the file server. Figure 1 shows an example benchmark testbed configuration with six LADDIS load generators distributed over two LAN segments, accessing an NFS file server. The LADDIS manager may be executed on a separate system, or on one of the

LADDIS load generators. Actual LADDIS measurements can be made using a wide range of load generator and network configurations.

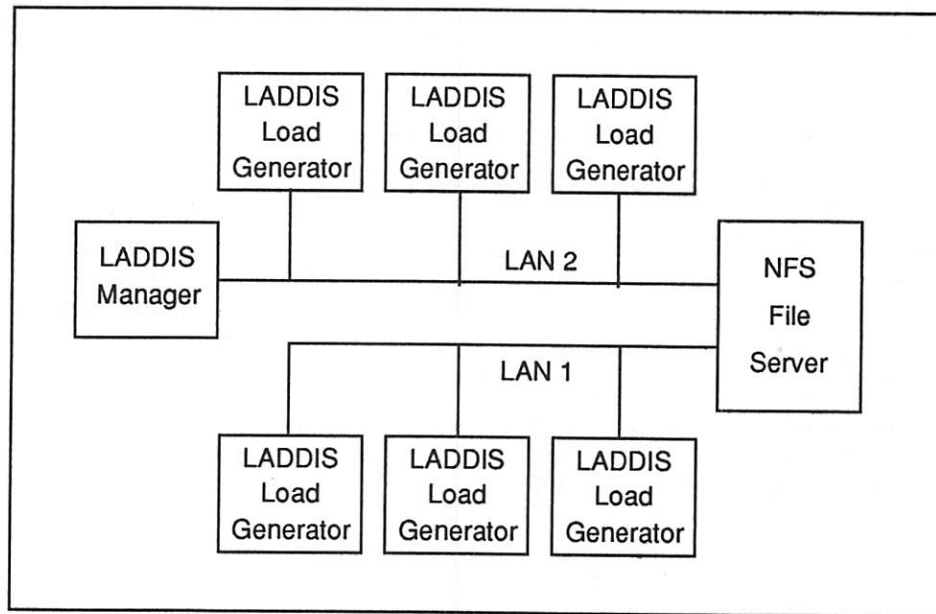


Figure 1. LADDIS Testbed Scenario

2.1. LADDIS Components

LADDIS executes as a collection of distributed, cooperating processes. A master shell script (`laddis_mgr`) reads the benchmark input parameter file (`laddis_rc`), and then spawns a remote shell script (`laddis_mcr`) on each of the load generators participating in the benchmark run. These shell scripts handle benchmark output, create log and error files, and spawn the various distributed processes involved in executing the benchmark.

The central LADDIS manager process (`laddis_prime`) controls the various phases of the benchmark and consolidates performance results from each LADDIS load generator. LADDIS synchronization daemons (`laddis_syncd`) execute on each load generator system and use an RPC-based message protocol to synchronize load generators while the benchmark is running.

During the load generation phase of the benchmark, multiple load generating processes (LADDIS) executing on each load generator send NFS requests to the file server. A parent process provides local control over the LADDIS load generating processes. The parent communicates with the synchronization daemon and central LADDIS manager to synchronize remote execution and report results. Figure 2 shows the communication paths of the LADDIS processes that execute on each load generating system.

2.2. User Interface

LADDIS includes an ASCII-based, menu-driven, user interface for managing benchmark configuration files, executing test runs, and viewing results files. The interface is similar to other SPEC benchmark suites [Dronamraju93].

2.3. Protocol Requirements

LADDIS implements the NFS protocol within the benchmark. This design enables the benchmark to maintain an accurate count of NFS requests sent to the file server. Correctness of the server's NFS implementation is verified during each benchmark run by a validation phase before beginning performance measurements.

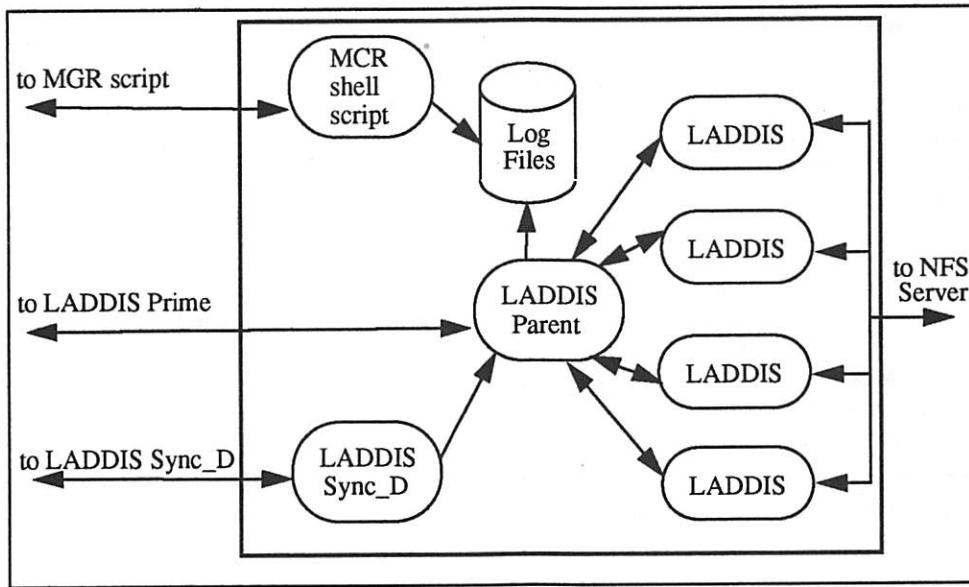


Figure 2. LADDIS Load Generator Processes

LADDIS uses standard ONC/RPC library interfaces, which provide the base of portability for the benchmark. The RPC library isolates LADDIS from the network transport layer implementation. To make use of asynchronous RPC calls, some standard RPC library functions have been re-implemented within LADDIS.

LADDIS uses standard TCP/IP transports for internal communications between distributed LADDIS processes and UDP/IP for sending requests to the NFS file server. SPEC SFS Release 1.0 Run and Report Rules require the use of UDP checksums on all network packets generated and received during benchmark execution.

3. Testbed Parameters

A LADDIS test run consists of a series of benchmark executions, each at increasing NFS load levels, generated by a set of LADDIS load generators and targeted at the NFS file server being tested. Most benchmark configuration parameters have a broad range of valid settings, but once chosen, they must remain constant across the LADDIS test run. The configuration parameters that are fixed across a test run are:

- all aspects of the server software and hardware configuration
- the number and type of LADDIS load generators used to create NFS load
- the number and type of network segments connecting the load generators to the server
- the arrangement of the load generators on the network segments
- the number of LADDIS load generating processes executing on each load generator
- all LADDIS workload parameters except the requested load level

3.1. Server Configuration

SPEC SFS Release 1.0 Run and Report Rules specify service characteristics that the NFS server is required to support (e.g., correctness of operations and reliability of data). NFS Version 2 protocol conformance is verified during the test run, just before beginning the series of performance measurements.

During test execution, LADDIS is not concerned with the details of the server configuration. However, once chosen, the server configuration must remain constant across the entire LADDIS test run. All relevant server hardware and software configuration information is reported with the performance results to ensure reproducibility.

3.2. Client Configuration

LADDIS can be run using one or more NFS load generator systems. The load generators need not be equipped identically, be the same model, or even be from the same vendor. The only requirement is to have enough load generators to saturate the file server's ability to service NFS requests. SPEC SFS Release 1.0 Run and Report Rules require that at least two NFS load generating systems and at least eight load generating processes be configured on each network segment. However, throughput saturation of an NFS file server may require a greater number of load generators, each producing many NFS requests concurrently.

A primary LADDIS design goal was to reduce client sensitivity in NFS performance measurement, and LADDIS has achieved that goal to a significant degree. Typically, each load generator system executes multiple LADDIS load generating processes (the LADDIS parent and synchronization daemon present minimal overhead during the performance measurement phase of the benchmark). Because these processes share the client system's CPU, memory, network device, and kernel software resources, the timesharing efficiency of these system elements can impact LADDIS response time measurements. Reducing the number of load generating processes can improve NFS response time. However, reducing the number of load generating processes may also reduce the client system's ability to generate load. If so, then additional load generator systems will be required to saturate the file server.

In addition to the cost of timesharing between processes, the load generating platform introduces some overhead into the measurement of server response time. The CPU speed and efficiency of the network controller affect the measurement. In addition, the efficiency of the platform's compiler and RPC library code may influence response time results. LADDIS load generation algorithms have been designed to reduce client platform sensitivity.

The sensitivity of LADDIS to client-based factors could be eliminated entirely by requiring all LADDIS performance measurements to be made using the same type of client system. This approach was rejected since few NFS environments are equipped with identical client configurations. Rather, SPEC advises that fast, efficient load generators be used when reporting LADDIS results. SPEC experience indicates that by employing powerful NFS client platforms as load generators, the effects of client sensitivity on results are minimized.

Before starting a LADDIS test run, the load generating capacity of each client system type should be determined [Watson92]. This can be done by running LADDIS between a single load generator system and the NFS server in isolation at increasing load levels until a maximum throughput is reached. The number of load generating processes can be adjusted, and response time and throughput results can be compared. The default value of four processes is a good starting point. This procedure should yield an upper bound on the load generator's ability to produce NFS load (relative to the targeted server) and determine the appropriate number of load generating processes to run on the load generating platform. Load generators should be added to the testbed configuration until the NFS server's saturation point is determined; this will ensure that LADDIS is measuring server saturation, rather than the NFS request limit of the load generators.

3.3. Network Configuration

LADDIS can be run using one or more network segments. The networks do not need to be the same type. The goal is to provide enough network bandwidth to allow the NFS load generators to saturate the server's ability to service NFS requests. However, depending on the NFS file server, simply adding additional load generators to a single network may result in measuring the network bandwidth rather than the NFS file server's throughput capacity.

An NFS file server may require more load than a single network can supply, regardless of the number of LADDIS load generators connected to it. Two questions must be answered: how many load generators are required to saturate a network segment, and how many network segments are required to saturate the NFS file server. Load generators can be added to a network segment until either the server or the network segment is saturated. Determining which is the bottleneck may require independent measurements of the network and server. For instance, a network analyzer can determine the network utilization level, and system utilities on the server, such as `sar(1)`, or `iostat(1)`, can measure the critical components of server performance. Experience within SPEC indicates that a single Ethernet⁵ is capable of supporting up to approximately 300 SPECnfs_A93³ operations/second.

Adding LADDIS load generators to a network segment may increase contention for network resources. Additional load generators may, in turn, introduce additional network collisions and retransmissions, which can

affect the response time measured by LADDIS. Some network contention is normal. SPEC SFS Release 1.0 Run and Report Rules require that at least two load generators be configured on each network segment. Reducing the number of load generators will reduce contention and may improve NFS response time. However, reducing the number of load generators might not make efficient use of the network segment and may introduce the need for additional network segments to saturate the server. In turn, additional network interfaces will be required on the server, and these may have associated performance costs on the server. The common practice is to configure enough load generators to saturate each network segment and no more.

3.4. Load Level

Having determined the appropriate setting for the benchmark testbed parameters, a series of LADDIS executions is run at increasing NFS request load levels. The LADDIS manager process distributes the load equally among the load generating systems, which, in turn, distributes the load equally among that system's load generating processes. Subsequent executions of the benchmark increase the load level. Eventually, the NFS file server becomes completely saturated with requests, and the measured load level no longer increases with additional requested load. Requesting additional load beyond saturation may produce a rapid increase in server response time, or may result in reduced NFS throughput on the server.

Each execution of LADDIS produces an average response time versus load level pair, and these are combined to form a LADDIS performance graph. The graph plots average server response time (on the y-axis) versus NFS request throughput (on the x-axis). The resulting response time/throughput curve has a number of interesting features. Figure 3 shows an example LADDIS performance curve. The general shape of the curve is flat or slowly rising followed by a steeper rise at higher load levels. The 50-millisecond response time level serves as an arbitrary reference for maximum average RPC response time. This level is sufficient for most NFS environments. The SPEC single-figure of merit performance value is derived from server throughput at 50 milliseconds average response time.

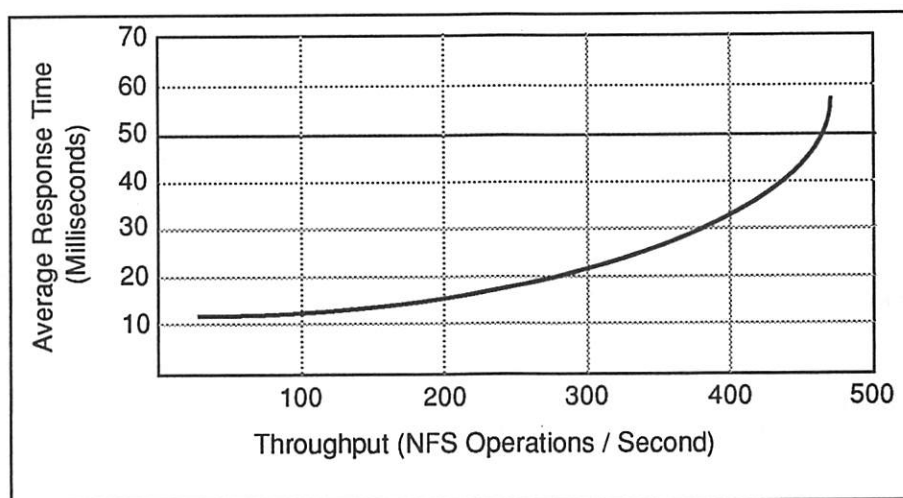


Figure 3. LADDIS Performance Graph

4. Workload Parameters

The goal of the LADDIS workload is to produce a synthetic approximation of an intensive software development environment. A number of workload studies have been applied, however no single study had sufficient breadth to establish all LADDIS parameter values simultaneously. Thus, a number of compromises, rules of thumb, and heuristics have been applied.

4.1. Operation Mix

LADDIS supports all operation types defined by the NFS Version 2 protocol. The default LADDIS operation mix is the same as the operation mix used in the nhfsstone benchmark. The mix is based on unpublished NFS client workload studies performed at Sun Microsystems during 1987 [Lyon92]. Standard LADDIS parameters specify an operation mix consisting of about half file name and attribute operations (LOOKUP 34% and GETATTR 13%), roughly one third I/O operations (READ 22% and WRITE 15%), with the remaining one-sixth spread among six other operations. Figure 4 presents the default operation mix percentages.

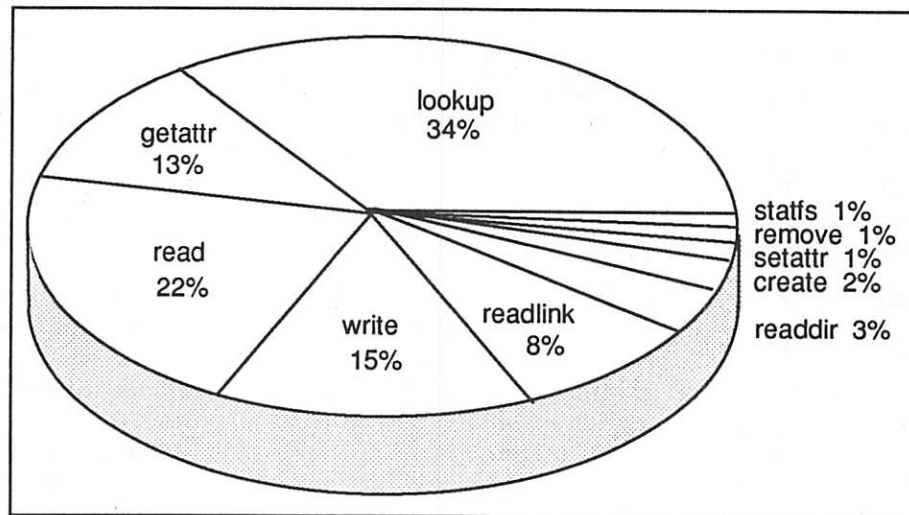


Figure 4. NFS Operation Mix

4.2. File Set

The set of test files accessed by LADDIS operations consists of a number of regular data files that scales with the requested load level, and a small fixed set of directories and symbolic links used for directory and symbolic link operations.

The test files (and the data they contain) are partitioned equally among all of the LADDIS load generating processes; each process populates and accesses its own subdirectory of files on the NFS file server. There are no shared files, and hence, no file access contention between load generating processes. For most applications, lack of contention is a reasonable assumption and has little bearing on overall NFS file server performance. For applications that do access shared data files and directories, the major performance impact is on client-based cache consistency policies which cause additional requests to be sent to the server. Because LADDIS is modeled on the operation mix actually delivered to the server, client cache hit rates do not affect the performance model.

Within each LADDIS process' private subdirectory, the test data files exist in a single, flat directory structure. One large directory per load generating process is unrealistic for most NFS file server environments where files are generally distributed throughout hierarchical directory trees. Normally, creating a large number of files in a single shared directory can cause a greater number of directory LOOKUP requests to be performed. Because LADDIS does not share its private subdirectory, there is no impact on the mix of operations. However, NFS server algorithms may be affected by the number of entries in each directory. Distribution of the test file population into subdirectory hierarchies would be more realistic. The effect of this change should be studied and possibly incorporated into a future version of LADDIS.

All LADDIS test file names are formed from a common initial sequence of nine characters followed by a three digit unique number, e.g. "dir_entry001." LADDIS limits file name length to provide support for UNIX⁶ systems that limit file names to 14 characters. The length limit and the similarity between file names may not be realistic for most NFS environments, and could affect the NFS server's file name storage algorithms. The LADDIS file name space could be improved by introducing some randomization into the file names.

4.2.1. File Set and Working Set

There are two fundamental properties of the LADDIS file set. First, a large, static *file set* is created. Second, a smaller *working set* of files is chosen from the larger file set. All NFS requests generated by LADDIS are targeted at the working set. The initial size of the file set and the growth and access characteristics of the working set are critical components of the LADDIS workload model.

LADDIS creates the file set during the benchmark initialization phase. The amount of data in the file set is scaled to the target load level specified for the benchmark run at the rate of 5 megabytes (MB) of data for each NFS operation/second (op/sec) of load. This scaling ensures that the NFS file server storage capacity scales with the NFS throughput offered by the server. The 5 MB per op/sec scaling factor recognizes the fact that NFS file servers store significant amounts of data, and should have the storage capacity to scale as larger and more active user populations are served. The choice of 5 MB per op/sec of load is meant to represent the static file set residing on today's medium to high-end NFS servers. For example, the file set for a server providing NFS throughput of 500 ops/sec would be initialized to about 2.5 gigabytes (GB); a 2000 ops/sec NFS server would have a 10 GB file set.

Each file is initialized and filled with 136 kilobytes (KB) of data. The file data is written to the file server during the benchmark initialization phase to ensure that storage space for the file is allocated. A file size of 136 KB results in approximately 40 files for each 5 MB of data and each NFS op/sec of requested load.

Using a uniform file size allows the number of files created by LADDIS to scale along with the data set size and server load level. Initializing each file to 136 KB provides a number of convenient properties:

- For UNIX operating system-based NFS file servers, a 136 KB file is large enough to ensure that the server allocates and references first-level, indirect, file index blocks.
- It allows a single read or write operation to produce a stream of up to 17 NFS 8-KB READ or WRITE transfers that access a contiguous file region, resulting in file accesses that provide for both single-threaded, spatial locality and multi-threaded, temporal locality (through BIOD simulation).
- It provides a wide range of starting file offsets for read and write operations.
- It simplifies file selection criteria, hastening the file selection process during the benchmark's performance measurement phase, thereby minimizing client sensitivity effects.

Despite the numerous advantages, a uniform file size is unrealistic. However, LADDIS file access length is not based directly on file size, and has little impact on the performance model. File size distribution could be revisited in a future LADDIS release, but probably is not necessary.

The working set of files is chosen randomly from the total file set. The working set provides the set of files that LADDIS accesses when performing NFS requests. There are a number of important advantages to using a working set. It models real-world environments where not all files stored on the NFS file server are accessed on a regular basis. It forces the NFS server to be configured to support more storage space than is needed to execute the benchmark. Also, by creating a file set and then choosing a random subset to be the working set, LADDIS ensures that the files being accessed are distributed within a larger set of files. Further, using a subset of files should create more realistic file caching and disk access patterns on the NFS file server.

The working set comprises 20% of the total file set. Thus, the amount of data in the working set and the number of files in the working set both scale with the requested load level at a rate of 1 MB per op/sec and 8 files per op/sec, respectively. No firm scientific basis exists for establishing the working set to be 20% of the total file set. Based on collective experience within SPEC, applying a 80/20 rule-of-thumb to the ratio of static to dynamic files seems appropriate.

4.2.2. Directories, Symbolic Links, and Non-working Set Files

For operations that create and remove files, LADDIS incorporates a number of simplifying design assumptions. First, LADDIS does not initiate operations that are expected to fail. For instance, remove operations are only performed on files that already exist. Total error avoidance is unrealistic, but greatly simplifies the description of what is being measured by LADDIS — successful NFS requests only. To avoid performing failed operations, LADDIS explicitly manages the file set name space and maintains existence state information for each file.

Directory and symbolic link operations are performed on a set of 20 directories and 20 symbolic links on each load generator, divided equally among the load generating processes. These fixed values are a gross simplification justified by the modest overall NFS performance impact of these operations. The NFS MKDIR, RMDIR, READDIR, SYMLINK, and READLINK requests together account for only 11% of the default LADDIS mix of operations. Although LADDIS provides parameters to set the number of directories and symbolic links, future versions should consider scaling these values along with the requested load level.

NFS REMOVE, RENAME and LINK requests, and most CREATE requests are performed on a small set of zero-length files that are not a part of the file working set. The number of these non-working set files is fixed at 100 on each load generator, divided equally among the load generating processes executing on the load generator. Maintaining a distinct group of non-working set files simplifies LADDIS data set size management by isolating requests that, as a side-effect, impact the number of bytes of data residing on the server. LINK and RENAME requests are restricted to non-working set files to maintain a strict separation of the working set and non-working set name spaces. The size of the non-working set is 100 files, half of which are initialized during the benchmark initialization phase. This value is adjustable via a benchmark parameter.

Normally, NFS CREATE requests are performed on the non-working set files, but they are also used to help manage the data set size.

4.3. Data Set Size

Given an initial working data set size of 8 136-KB files for each op/sec of requested load (or approximately 1 MB of data per op/sec), LADDIS controls the fluctuation in the data set size as the benchmark executes. Two factors are involved: file write append operations cause the data set to grow, and file truncation operations cause it to shrink.

LADDIS performs two distinct types of write operations: writes that overwrite existing data in a file, and writes that append new data to the end of a file. These operations exercise different parts of the server's file system implementation. Depending on data buffer alignment, over-writing data may require pre-fetching the old data before overwriting part of it, while appending new data may require allocating space from the file system's pool of free storage space. Default LADDIS parameters specify that 30% of write operations over-write existing data and 70% of write operations append new data. Appending new data increases the total amount of data in the file set. If unchecked, append operations would cause uncontrolled growth of the data set size as each benchmark execution proceeded.

LADDIS limits data set growth to a 10% increase. The limit is implemented by using some of the NFS SETATTR and CREATE requests specified by the operation mix to perform file truncations (by setting the file length argument to 0). These special truncation operations are performed whenever a write append operation would cause data set size growth to exceed 10%. The truncation is performed just before the append. These truncation operations are included in the calculation of the operation mix percentages for SETATTR and CREATE.

Using SETATTR and CREATE requests to perform file truncation models real-world, UNIX operating system-based open/truncate sequences, which are performed when updating a file from a temporary copy of the file. This approach produces a bi-modal distribution of response times for these operations; some SETATTR requests simply update a file's attributes, but others perform file storage space deallocation.

Recent research into file write access patterns [Hartman92] indicates that append ratios as high as 90 - 95% may be common in some environments. Using the default LADDIS settings, a stable data set size can not be maintained with a 90% append ratio - there are not enough SETATTR and CREATE requests to counter-balance the append operations. Maintaining a stable data set size depends on the relative percentages of WRITE, SETATTR, and CREATE requests in the operation mix, the average file size, the average length of an append operation, and the data set fluctuation limit. The 70% append ratio maintains a stable data set size within the default LADDIS parameters.

The values for the write append percentage and data set growth limit parameters are interdependent. The 10% fluctuation value was chosen to minimize the number of SETATTR and CREATE requests required to maintain a stable data set size. All file truncations shorten existing files to zero length for the same reason. There is a strong desire to have LADDIS maintain a stable data set size so results produced at different load levels or with different run times will access a known file set and can be compared.

4.4. Read and Write Operations

LADDIS read/write operations impact NFS performance. Depending on how read/write operations are monitored, different characteristics become apparent. Viewed from the network, LADDIS controls the amount of data transferred in each network packet. From the perspective of an individual file access, LADDIS controls the length of each byte stream to and from the file. From the NFS file server's point of view, LADDIS controls the aggregate data transfer rate to and from the server. A number of LADDIS parameters interact to control these characteristics.

4.4.1. Network Packet Size Distribution

The NFS Version 2 protocol allows up to 8 KB of file data to be transferred in each network transfer (lower level transport protocols may subdivide the NFS packet further). Most NFS client kernel implementations provide data caching and some amount of file read-ahead and write-behind activity. Read-ahead and write-behind allow network transfers to be performed independently of an application's access pattern, producing fewer network packets and larger transfer sizes. As a result, most NFS READ/WRITE requests are transferred in network packets containing 8 KB of file data. However, because file lengths are seldom an exact multiple of 8 KB, transfers accessing the end of a file typically contain less than 8 KB of data.

LADDIS controls the amount of data contained in each NFS read/write transfer to produce a known distribution of network data packet sizes. The packet size for any given request is chosen randomly, but the choice is weighted to produce a target packet-size distribution. The default LADDIS network packet distribution produces approximately 90% read packets containing 8 KB of data, and 10% packets containing less than 8 KB. Roughly 50% of the write packets contain 8 KB, and 50% contain less than 8 KB. The distribution of "fragment" packets containing less than 8 KB of data, is spread evenly across packets containing multiples of 1 KB. These values are derived from NFS packet transfers observed by one of the authors [Keith90]. The only simplification made by LADDIS is to limit fragment packet sizes to multiples of 1 KB and distribute them equally across 1 - 7 KB sizes, as shown in Figure 5.

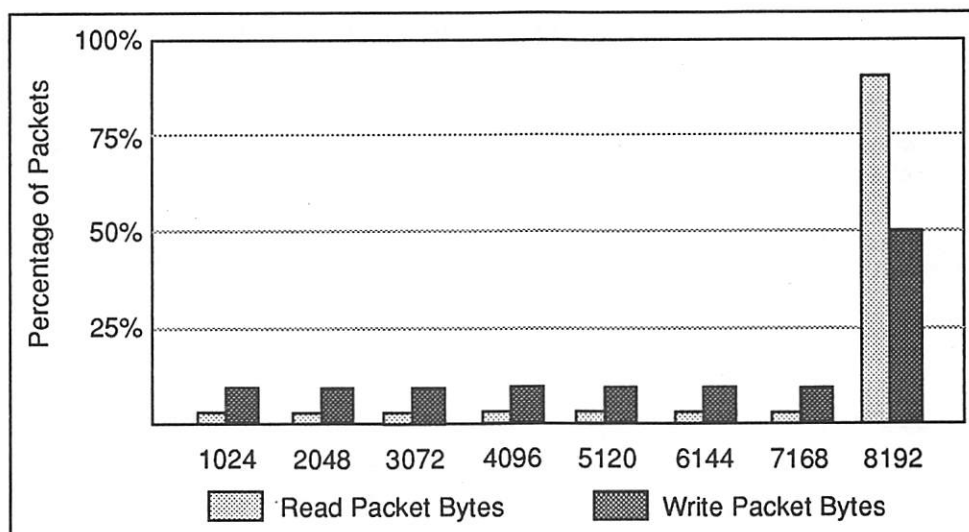


Figure 5. Read and Write Packet Size Distribution

4.4.2. File I/O Data Streams

LADDIS file accesses are modeled on byte stream operations that can be larger than the single NFS 8-KB packet size. High-level file read/write operations targeted at a single file can vary from 1 KB to 136 KB in length. The length of the operation is limited to the initial file size of 136 KB to ensure that most LADDIS test files can

successfully accommodate the largest possible read operation. This artificial limit helps to simplify file choice overhead during the benchmark performance measurement phase.

Of course, the actual access to the file server is implemented as a stream of NFS 8-KB READ/WRITE requests, possibly followed by a single 1 - 7 KB fragment request. The distribution of high-level file accesses is tailored to produce the required percentage of 8 KB and 1 - 7 KB fragment network packets. This distribution is achieved by weighting the choice of file access length to produce many short accesses of 1 - 15 KB and fewer long accesses of 16 - 135 KB [Baker91]. Beyond packet sizes of 15 KB, the distribution is essentially logarithmic (e.g., half as many 128 KB accesses as 64 KB accesses, etc.), although some variations involving fragment packets are needed to meet the packet size distribution requirements. See Figure 6.

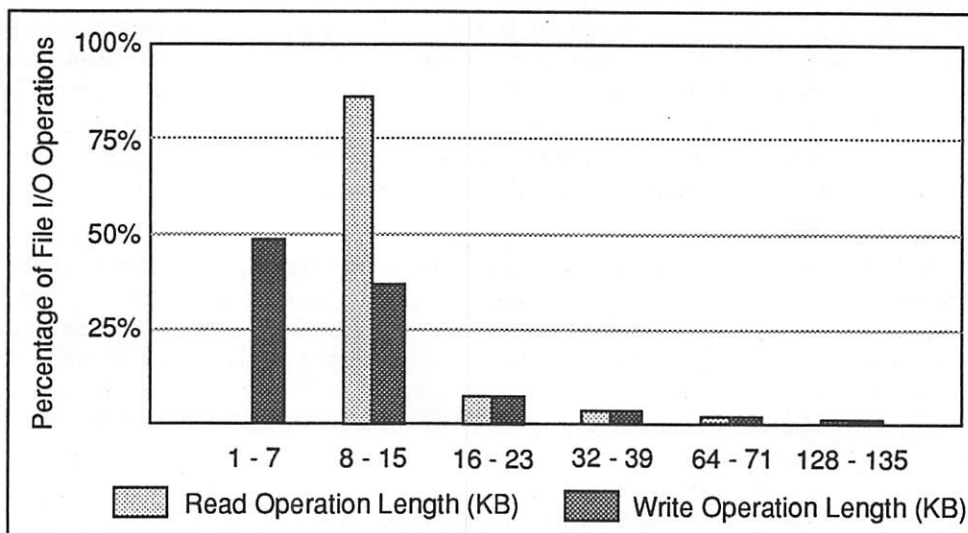


Figure 6. File I/O Operation Length Distribution

The starting file offset for each access is chosen to be an exact multiple of 8 KB. Because most NFS implementations perform client-side data caching, 8-KB boundaries are an obvious choice for file offsets in individual packet transfers. For read operations, the offset is chosen randomly from the set of file offsets that will allow the read operation to succeed without exceeding the current end-of-file. For write operations that over-write existing data, the starting offset is chosen as it is for reads. For write operations that append new data to end of the file, the current end-of-file offset is used as the starting offset.

This model of file access synthesizes client cache read-ahead and write-behind implementations and presents some spatial locality of file reference to the NFS server.

4.4.3. BIOD Simulation

LADDIS models temporal locality of file accesses by simulating NFS Block I/O Daemon (BIOD) file access patterns. The simulation is performed using asynchronous RPC requests for multi-packet file read/write operations.

Many NFS client implementations support BIOD threads of control which perform file read-ahead and write-behind operations between the client kernel's data cache and the NFS server. Normally, a number of BIOD processes are supported, allowing many READ/WRITE requests to be sent to the server concurrently before awaiting a reply to the first request. The default LADDIS BIOD setting requires that whenever a LADDIS load generating process performs a read/write operation longer than 8 KB, at least two NFS READ or WRITE requests are sent to the server before waiting for a reply.

This setting produces several effects on the NFS file server. First, because multiple READ/WRITE requests to the same file are sent to the server within a short time-span, the server may be able to provide service

efficiencies based on the file access locality. Second, the total number of requests submitted to the server concurrently is not limited to the number of LADDIS load generating processes. Rather, each LADDIS process may send multiple requests, generating a burst of increased load over a short period of time. Such a burst may introduce additional short-term stress on the file server's resources.

4.4.4. Aggregate Data Transfer Rate

A number of random factors affect the LADDIS data transfer rate at any given point during a test run. These factors include which operation is being performed and how much data is transferred by the read/write operations. The average data transfer rate can be calculated from the overall load rate, the percentage of READ/WRITE requests in the operation mix, and the average transfer size for each READ/WRITE request.

For the standard LADDIS distribution, the average transfer size for WRITE requests is 6 KB (50% 8-KB and 50% randomly distributed across 1 - 7 KB), and the size for READ requests is 7.6 KB (90% 8-KB and 10% randomly distributed over 1 - 7 KB). Thus, for a 100 ops/sec load rate, using the default operation mix of 15% WRITE requests and 22% READ requests, the data transfer rate is $100 \text{ ops/sec} * ((0.15 * 6 \text{ KB/op}) + (0.22 * 7.6 \text{ KB/op})) = 257 \text{ KB/sec}$.

5. Load Generation

LADDIS load generation consists of a paced stream of NFS requests, separated by random delays. Individual operations are chosen randomly but weighted according to the operation mix. The operation is performed on a randomly chosen file, with the choice weighted by a file access pattern distribution function, and constrained by the requirement to choose a file that will satisfy the operation successfully. Each NFS request is timed, and per-operation type statistics are maintained. Periodically, the load rate is adjusted as necessary.

5.1. Load Rate Management

In order for each LADDIS load generating process to produce NFS requests at the specified load rate, the server's average request response time is estimated periodically.

LADDIS employs a warm-up phase to calibrate the server's average response time before beginning the benchmark performance measurement phase. Every 2 seconds during the warm-up phase, each LADDIS process calculates an average inter-request delay period that allows it to produce the requested load rate. The calculation is based on the server's average response time over the previous 2 seconds. The warm-up phase executes for 60 seconds. The warm-up phase duration was established empirically to produce a steady state of request generation for load levels as low as 10 ops/sec for an individual load generating process.

At the end of the warm-up phase, all performance counters and statistics are reinitialized and further server recalibrations are performed at 10 second intervals. Between each operation, LADDIS pauses for a random time of from one half to one and a half times the current inter-request delay period before beginning the next operation.

When a shortfall or overage in load rate is observed, LADDIS aggressively adjusts the inter-request delay period to allow it to make up any cumulative request shortage or overage during the next measurement period. One effect of this algorithm is that during a test run where the requested load can not be achieved, LADDIS quickly eliminates the inter-request delay. For load levels near the file server's capacity, this is the desired behavior. For load levels beyond the server's capacity, LADDIS may tend to overburden the server and introduce CPU inefficiencies on the load generator by attempting to produce more load than the server can handle. In this case, response time may rise significantly as may the number of errors due to RPC time-outs.

5.2. Operation Choice

LADDIS randomly chooses the next operation to perform based on a probability derived from the operation mix. Some operations cause more than one NFS request to be sent to the server. For instance, write operations may involve a data length that causes multiple WRITE request packets to be sent to the server. Based on the packet size

distribution table, LADDIS calculates the average number of requests to be generated by each read/write operation and re-weights the operation mix accordingly.

By choosing operations randomly, LADDIS loses the ability to generate common NFS request sequences (e.g., a READDIR followed by a GETATTR, or a LOOKUP followed by a read). Operation context sensitivity would be a nice addition to a future version of the benchmark.

5.3. File Choice

File choice depends on the selected operation type. For directory and symbolic link operations a file is chosen randomly from the small set of files supporting that operation. The choice is constrained only by the existence attribute of the file, so that the operation will succeed. Similarly, file creation, removal, and naming operations access a random file chosen from the non-working set files, according to the same requirements.

If an appropriate file can not be found (e.g., no directories currently exist but a RMDIR request is to be performed), then LADDIS randomly chooses another operation to perform. An operation that can not be performed immediately due to lack of an appropriate file may be successful later. If too many operations can not be performed for this reason, then the test run will not achieve the target operation mix and will be flagged as invalid. This can result from specifying an operation mix that the benchmark can not produce, for instance, specifying a very high percentage of remove operations and a very low percentage of create operations.

The remaining operations choose files from the static file working set. Files selected for read/write operations must be long enough to accommodate the operation. The length of each read/write operation is chosen randomly, and weighted according to the file access length distribution. For read and over-write operations, the file chosen must be long enough for the read to complete successfully.

5.4. File Access Patterns

When choosing a file from the working set to be accessed, LADDIS implements a non-uniform, file access distribution algorithm. During the benchmark initialization phase, the working set is partitioned into small groups of files, and each group is assigned a probability of providing the next file to be accessed. The probabilities are based on a Poisson distribution. Each group contains the same number of files, and the files within each group have an equal chance of being selected (constrained by the file length requirements for read and over-write operations).

As the benchmark load level is increased during successive test runs, additional files are created in the working set, and the number of file groups is increased. Scaling the number of file groups with the load level adds more values to the Poisson distribution and smooths the file access distribution curve. The Poisson distribution is applied to groups of files rather than individual files to improve the chances that a file selected for a read/write operation will be long enough to support the operation.

LADDIS employs a non-uniform, file access distribution to increase file access locality. The desired effect is to increase file server buffer cache hit rates. Other non-uniform mechanisms have been suggested to increase locality further, including Markov processes. It is thought that file access choice based on recent file access patterns might provide a more realistic model of file access. The implementation of the LADDIS file selection algorithm has been isolated from the main code path to encourage future experimentation.

5.5. Response Time Measurements

After selecting an operation and a file, LADDIS sends the request or series of requests to the server. The elapsed time of sending each NFS request and waiting to receive a reply is the basis for determining the server's response time. The measurement includes time spent executing in the local RPC library and network protocol code path while sending and receiving packets, processing time required to determine the elapsed time, and any additional client platform operating system overhead or interference from other LADDIS processes executing concurrently on the load generator system. Asynchronous RPC calls are handled similarly, but include a small amount of extra processing required to manage multiple concurrent requests.

The accuracy of each individual measurement depends on the granularity of the client's `gettimeofday(1)` implementation. Assuming that the error in each measurement is random, an average value derived from a large

number of sample measurements produces an unbiased estimate of the actual average response time. For each operation type, the benchmark reports a 95% confidence interval for the mean response time reported.

Only successful requests are included in these results. Failed requests are tallied separately, but are not included in response time measurements or in the server's throughput calculation. Except for rare, unexpected errors reported by the server, failed requests indicate RPC time-out errors. LADDIS uses a single, non-adaptive time-out mechanism to define the maximum allowable service time for each request. This is a simpler mechanism than is used by most NFS implementations, but reduces the overhead required to generate NFS load. Three RPC time-out classes are defined, and no requests are retried. The three classes are for 1, 2, and 3 seconds and are applied to file attribute read requests, file data read requests, and file data and attribute write requests, respectively.

5.6. Test Duration

The performance measurement phase of each benchmark execution (each performance graph data point) lasts for 10 minutes. The time period was established empirically as the minimum time required for each LADDIS load generating process to achieve a close approximation of the operation mix, when generating requests at the rate of 10 ops/sec. Normally, test runs are conducted at higher load rates with multiple processes and they achieve an aggregate operation mix approaching the target mix in only a few minutes.

The duration of the test run can significantly impact the amount of file data that becomes cached in the file server's data buffer caches. If a large percentage of the data set can be accommodated in the server's buffer cache, longer warm-up periods and longer test runs may produce better results. Therefore, only results from test runs of the same duration are comparable.

5.7. Algorithm Trade-offs

Each step in the LADDIS load generation algorithms involves trade-offs between implementing more realistic load generation behavior and reducing the internal overhead required to produce NFS requests. For instance, operation choice could be made sensitive to the recent operations, but this would require additional algorithmic complexity to achieve the desired mix of operations. Non-uniform file access could be applied to symbolic link operations with only a minor increase in processing time, but, for the default LADDIS parameter settings, would have a negligible impact on server performance. RPC time-out handling could be improved, but would complicate reporting procedures.

Most of these issues could be addressed by increasing the overhead involved in generating requests and would reduce LADDIS's load generating capacity. Extra load generators could be required to measure an NFS server. Also, executing additional code within LADDIS introduces additional client sensitivity. LADDIS attempts to balance the goals of realistic load generation and increased load generating capacity by focusing on the areas with the greatest impact on server performance and simplifying the approach in less critical areas.

6. Benchmark Results

Each execution of the LADDIS benchmark produces a detailed results report for each load generator, and an aggregate results report combining the results from all load generators involved in the test run. Each report includes detailed information for each NFS operation type, parameter setting and file set information, and a summary description of server throughput and average response time, as shown in Figure 7. If more than 1% of the NFS requests failed, including timed-out requests, LADDIS indicates that the test run is invalid.

The throughput and average response time reported by individual test runs are combined into a LADDIS performance graph showing the server's response times over a range of load levels. This graph and all LADDIS configuration information is combined on a standardized, 2-page, SPEC SFS Release 1.0 performance report. This is the highly recommended way to judge overall file server performance results. Results may be reported as "baseline" or "non-baseline" results. Baseline results are for server configurations that conform to all standard NFS service requirements defined by the NFS Version 2 Protocol and the SPEC SFS 1.0 Run and Report Rules [SPEC93].

Aggregate Test Parameters:

Number of processes = 36
 Requested Load (NFS operations/second) = 240
 Maximum number of outstanding biod writes = 2
 Maximum number of outstanding biod reads = 2
 Warm-up time (seconds) = 60
 Run time (seconds) = 600
 File Set = 9612 Files created for I/O operations
 1908 Files accessed for I/O operations
 (each prefilled to 136 KB)
 612 Files for non-I/O operations
 144 Symlinks
 144 Directories
 Additional non-I/O files created as necessary

LADDIS Aggregate Results for 6 Client(s), Fri Mar 12 18:54:07 1993
 LADDIS NFS Benchmark Version 0.1.22, Creation - 10th December 1992

NFS Op Type	Target NFS Mix Pcnt	Actual NFS Mix Pcnt	NFS Op Success Count	NFS Op Error Count	Mean Response Time Msec/Op	Std Dev Response Time Msec/Op	Std Error of Mean, 95% Confidence +- Msec/Op	Pcnt of Total Time
null	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
getattr	13%	13.1%	18962	0	5.71	5.51	0.03	2.7%
setattr	1%	0.8%	1293	0	34.52	35.71	0.33	1.1%
root	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
lookup	34%	33.6%	48504	0	7.35	7.59	0.02	9.0%
readlink	8%	8.1%	11815	0	5.73	5.56	0.04	1.7%
read	22%	21.7%	31410	0	15.73	18.05	0.05	12.5%
wrcache	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
write	15%	15.5%	22430	0	116.36	127.54	0.15	66.4%
create	2%	1.9%	2857	0	41.03	49.43	0.26	3.0%
remove	1%	0.9%	1350	0	11.88	13.30	0.19	0.3%
rename	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
link	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
symlink	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
mkdir	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
rmdir	0%	0.0%	0	0	0.00	0.00	0.00	0.0%
readdir	3%	2.9%	4243	0	25.15	23.06	0.14	2.7%
fsstat	1%	0.9%	1422	0	5.88	6.34	0.13	0.2%

| LADDIS VERSION 0.1.22 AGGREGATE RESULTS SUMMARY |

NFS THROUGHPUT: 240.45 Ops/Sec AVG. RESPONSE TIME: 27.23 Msec/Op
 NFS MIXFILE: [LADDIS default]
 AGGREGATE REQUESTED LOAD: 240 Ops/Sec
 TOTAL NFS OPERATIONS: 144286 TEST TIME: 600 Sec
 NUMBER OF LADDIS CLIENTS: 6
 TOTAL FILE SET SIZE CREATED: 1307232 KB
 TOTAL FILE SET SIZE ACCESSED: 259488 - 285390 KB (100% to 109% of Base)

Figure 7. Detailed Aggregate Results Report

7. Areas for Future Work

LADDIS provides a rich set of workload parameter settings that could be used to create workload abstractions for CASE, CAD, office automation, and other common NFS environments. Second-order parameter sets

could be created to model combinations of these application environments. Workload studies are needed to establish the basis for these abstractions. The current default LADDIS workload was derived from a variety of research studies, anecdotal evidence, rules of thumb, and common sense. Future workload characterizations should encompass all key factors affecting NFS server performance within a single study. These factors include NFS operation mix, network packet characteristics, file set size and distribution by file type, data set size and distribution within the file set, working set size, and file access patterns.

The implementation of LADDIS lends itself to adaptation to other benchmark situations. For instance, isolating the actual formulation of NFS RPC request packets within separate subroutines facilitates the creation of a new LADDIS implementation supporting future releases of the NFS protocol. Additionally, the LADDIS workload parameterization and load generation framework can be applied to other distributed file systems such as DCE/DFS. Changing LADDIS to produce local operating system calls (like `nhfsstone`) instead of network requests (like LADDIS) would greatly impact client sensitivity; however, the existing LADDIS load generation algorithms and workload parameterization would probably still be applicable.

The fundamental LADDIS design goal of reducing client sensitivity has been achieved through a number of design decisions. SPEC experience indicates that using fast, efficient NFS client platforms as LADDIS load generators minimizes the client platform's impact on performance measurements. However, as LADDIS is deployed throughout the industry and additional experience is gained on a broad set of client platforms, there will be an increased need for careful quantification of LADDIS client sensitivity effects.

There are also a number of technical improvements that could be made to LADDIS including:

- shared files and directories, among multiple LADDIS processes,
- test files distributed across a hierarchical directory structure,
- randomization factors included in file name generation,
- variable file data lengths,
- the number of directories, symbolic links, and non-working set files scaled with load level,
- improved RPC time-out and error mechanisms,
- separate response time reporting for truncation operations,
- operation choices sensitive to recent request streams,
- additional file access distribution mechanisms.

However, the specific benefit that each of these changes might introduce into the performance measurement must be weighed carefully against client sensitivity effects introduced by the algorithmic changes. Further, the potential impact that the change will have on the benchmark's accuracy and ability to produce the desired workload abstraction must be considered.

8. Acknowledgments

The LADDIS benchmark is the result of the efforts of many individuals over the past three years: Ken Teelucksingh (Digital) provided the multi-client support, John Corbin (Sun) contributed the asynchronous RPC code, Santa Wiryaman (Digital) wrote the validation suite, and Richard Bean (Data General) did the original port from `nhfsstone`.

The LADDIS Group was critical to the technical evolution of the benchmark: Bruce Nelson (Auspex) inspired and chaired the LADDIS effort, Bob Lyon (Legato) provided an important historical perspective, Maneesh Dhir (Sun) and Peter Olenick (Interphase) debugged and characterized early versions of the benchmark, Brian Pawlowski (Sun) contributed significant technical input, Janet Johnson (NCSU) helped out with statistical analysis, and a number of others discussed requirement and design issues over the course of two years.

The SPEC SFS subcommittee played a crucial role in turning the benchmark into a product: Krishna Dronamraju (AT&T/NCR) and Jeff Reilly (Intel) integrated the SPEC tools and user interface, Tom Spuhler (HP), Rick Jones (HP), and Andy Watson (Auspex) were key contributors to workload discussions, the SPEC membership ported LADDIS to a broad range of platforms, and the SPEC steering committee and board members helped resolve a number of key issues and focused efforts toward releasing the benchmark.

9. Author Information

Mark Wittle is a Principal Software Engineer in the DG/UX⁷ Development Group at Data General Corporation. Mark has been the primary technical contributor to the design and implementation of LADDIS load generation functionality. Mark holds a bachelors degree in Mathematics from Knox College in Galesburg, IL, and attended the University of Illinois graduate Computer Science program for two years. He can be reached at wittle@dg-rtp.dg.com, or, Data General Corp, 62 Alexander Drive MS-109, Research Triangle Park., NC 27709.

Bruce Keith is the DEC OSF/1⁸ AXP⁸ NFS performance project leader in the Systems Engineering Characterization Group at Digital Equipment Corporation. He is the chairman of the SPEC SFS subcommittee and is the technical project leader for LADDIS and SFS release manager within SPEC. Bruce received his B. S. degree in Computer Science from Worcester Polytechnic Institute. He can be reached at keith@oldtmr.enet.dec.com, or, Digital Equipment Corporation, 85 Swanson Road BXB1-1/F11, Boxboro, MA 01719.

References

- [Baker91] Baker, Mary G., et al., "Measurement of a Distributed File System," Proceedings of the 13th Symposium on Operating System Principles, pp. 198 - 212, October 1991.
- [Dronamraju93] Dronamraju, S. Krishna, et al., "Managing the SFS User Interface," SPEC Newsletter, Volume 5, Issue 1, pp. 5 - 10, March 1993.
- [Hartman92] Hartman, John, "File Append vs. Overwrite in a Sprite Cluster," Sprite Project, University of California at Berkeley, Presentation to the LADDIS Group on January 21, 1992.
- [Keith90] Keith, Bruce, "Perspectives on NFS File Server Performance Characterization," Proceedings of the Summer 1990 USENIX Conference, pp. 267-277, June 1990.
- [Legato89] "nhfsstone" NFS load generating program, Legato Systems Inc., Palo Alto, CA 94306
- [Lyon92] Lyon, Robert, private communication to the LADDIS group, June 26, 1992.
- [Sandberg85] Sandberg, Russel, et al., "Design and Implementation of the Sun Network File System," Proceedings of the Summer 1985 USENIX Conference, pp. 119-130, June 1985.
- [Shein89] Shein, Barry, et al., "NFSSTONE - A Network File Server Performance Benchmark," Proceedings of the Summer 1989 USENIX Conference, pp. 269-274, June 1989.
- [SPEC93] Standard Performance Evaluation Corporation, "SPEC SFS Release 1.0 Run and Report Rules," SPEC SFS 1.0 Release, SPEC c/o NCGA, 2722 Merrilee Drive, Suite 200, Fairfax, VA 22031-4499
- [Stern92] Stern, Hal, et al., "NFS Performance and Network Loading," Proceedings of the LISA VI Conference, pp. 33 - 38, October 1992.
- [Watson92] Watson, Andy, et al., "LADDIS: Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark," Proceedings of the LISA VI Conference, pp. 17-32, October 1992.

Trademarks

- ¹NFS and ONC are trademarks of Sun Microsystems, Inc.
- ²Nhfsstone source code is a copyrighted product of Legato Systems, Inc.
- ³SPEC and SPECnfs_A93 are trademarks of the Standard Performance Evaluation Corporation.
- ⁴OSF/1 is a registered trademark of Open Software Foundation, Inc.
- ⁵Ethernet is a trademark of Xerox Corporation.
- ⁶UNIX is a registered trademark of Unix Systems Laboratories.
- ⁷DG/UX is a trademark of Data General Corporation.
- ⁸DEC OSF/1 and AXP are trademarks of Digital Equipment Corporation.

Design and Implementation of a Multimedia Protocol Suite in a BSD Unix Kernel*

Lakshman K, Giri Kuthethoor, Raj Yavatkar

Department of Computer Science
University of Kentucky, Lexington, KY 40506

Abstract

Development of distributed multimedia applications requires support for coordination and temporal/causal synchronization of traffic over related streams. Our current research involves investigation of appropriate OS and communication abstractions to support such applications. Towards this goal, we have designed and implemented MCP, a suite of transport and session layer protocols, in the framework of a standard BSD Unix networking platform. MCP contains two new abstractions. First, MCP contains a token-based mechanism for coordination of traffic over a multipoint connection. Second, MCP includes an abstraction called a *multi-flow conversation* that enforces both temporal and causal synchronization among related data streams. This paper discusses Unix kernel implementation of MCP and describes our experience in using MCP.

1 Introduction

With the increasing use of high speed networks, it is now possible to build multimedia applications that involve geographically dispersed group of users. These applications involve concurrent communication of text, voice, graphics, and video. Our current research involves investigation of appropriate OS and communication abstractions to support distributed multimedia applications. Towards this goal, we have designed a suite of transport and session layer protocols [Yav92] and implemented the protocols in the framework of a standard BSD Unix networking platform.

This paper describes our experience in design and implementation of transport and session layer protocols for distributed multimedia applications.

Development of distributed multimedia applications requires resolution of several issues including media coordination and synchronization, media mixing, OS scheduling, and the problem of network resource allocation to provide performance guarantees in terms of delay bounds and bandwidth reservation. However, we must emphasize that this paper only addresses the question of providing suitable transport and upper level protocol abstractions to meet the temporal/causal synchronization and coordination needs of such applications. In particular, the paper addresses the following aspects of building distributed multimedia systems:

1. Unique communication requirements posed by distributed multimedia applications and need for appropriate communication support to facilitate development of such applications.
2. Design and implementation of suitable transport and session layer communication mechanisms using the SUNOS 4.1.1 version of Unix as the experimental platform.
3. Investigation of suitability and extensibility of the UNIX IPC facilities, protocol software structure, and user interface in accommodating needs of multimedia applications.
4. User interface requirements to support multipoint, collaborative applications.

*This research is supported in part by the National Science Foundation Grant No. NCR-9111323 and Grant no. STI-9108764.

Rest of this paper is organized as follows. Sections 2 and 3 provide the motivation and background for our work. Section 4 describes our approach including design of MCP and discusses the design alternatives and tradeoffs. Section 5 describes our implementation of MCP suite in SunOS 4.1.1 kernel. We have conducted a systematic performance evaluation of MCP kernel implementation and have also compared the performance of user applications with and without MCP. Sections 6 and 7 discuss the results of our performance evaluation, Section 8 compared our work with work by others, and Section 9 provides a summary of our work.

2 Motivation

Our goal is to facilitate development of multimedia distributed applications that involve a geographically dispersed group of users. An example of such an application is a collaborative, software engineering environment [EGR89, SFB⁺87]. In such an environment, a group of designers located at different sites collaborate on a design document (or a program) using interactive tools to edit and test parts of the design under development. Interaction may involve a group editor (based on shared windows), an image display (that displays resulting design), and a voice channel that allows them to view, discuss, and edit the suggestions made by each other.

Our main focus here is on the issues of coordination and synchronization of traffic over related data streams. These issues arise in following forms:

- A single multipoint communication involving multiple users on multiple, remote sites requires causal synchronization so that all participants “see” all the communication events in the same or “correct” order. For instance, a voice conference channel that spans multiple participants requires such a synchronization.

Apart from voice/video interactions, both communication and application-level software must also enforce some degree of concurrency control when a group of users may simultaneously view and update shared text or image data to maintain consistency.

- The communication software must allow an application to coordinate multiple information channels that carry traffic from multiple media such as text, voice, video (or image). The collaborative, software development group effort described earlier is an example of such an interaction. For such an application, the communication software must allow related streams to be grouped together and recognize the order and sequencing of traffic sent over them. For example, when a participant scrolls through an image browser (or a shared editor window) and says, “look at the middle of the display”, the statement should be heard at the same time (or just after) the scrolling is completed. Such temporal relationships must be captured in the delivery of traffic over related streams. Another example of such a synchronization is “lip-synching” when voice and video are transmitted over separate connections in a digital network.

Existing transport and/or session layer protocols lack communication abstractions that provide the necessary semantics for coordination and temporal synchronization in multimedia applications. We are currently investigating such abstractions in our research and are using them in building **PolySchmues**, a distributed interactive environment for collaboration using multiple media.

In the following, we first elaborate on the problem of coordination and temporal synchronization and then discuss proposed solution.

3 Coordination and Temporal Synchronization Problem

Hereafter, we will refer to a single or a multipoint connection as a *flow*. We assume that the network layer provides a flow abstraction that may have performance guarantees associated with it to provide predictable performance needed by real-time voice or video channels [FV90]. The question of how to satisfy the real-time performance requirements of a flow is *not* the focus of discussion here; that question has been addressed by many researchers [FV90, ZDE⁺93, Yav89]. Instead, the focus here is on research issues in using and combining such flows to build multimedia, collaborative, distributed applications. For the design of multimedia systems such as PolySchmues, one must address two aspects of coordination, namely, what sort of coordination is

necessary and how much control must be exercised at the communication substrate. In our view, two kinds of coordination are necessary:

Intra-Flow Coordination Given a flow spanning a group of users, one must consider the problem of concurrency control when more than one user may be transmitting data at the same time. A common example of such coordination is avoiding interruptions and crosstalk in a multipoint voice channel. Such coordination is also necessary when communication involves a shared text document or an image.

Inter-Flow Coordination A multimedia application may need to synchronize traffic over multiple flows, each carrying traffic from a different medium (text, voice, video, or image). A collaborative, software development group effort described earlier is an example of such an interaction.

3.1 Degree Of Coordination

The amount of coordination needed varies in both inter- and intra-flow cases depending on the application.

In the case of a shared window package [SFB⁺87], a single window is displayed on the display screens of multiple users, and each user gets an identical image of the window. Shared window systems provide no concurrency control for simultaneous actions by multiple users. Instead, they allow users to constantly see the actions of other users who are responsible for manually ensuring that there are no conflicts.

At the other extreme lie teleconferencing systems in which interference is avoided by strict control based on the notion of a "floor", where only the current "speaker" that has the floor is allowed to "speak" (or transmit data) at any time. Strict coordination is also necessary among separate flows when a group of users are pointing at a shared text window and discussing parts of text over a voice channel.

Both the extremes have their limitations. On one hand, lack of any concurrency control puts additional responsibility on application designers to resolve conflicts. On the other hand, strict floor-based control does not allow applications to exploit inherent concurrency and may sometimes unnecessarily increase latency due to the waiting involved. In addition, there is a continuum between these two extremes where varying degree of coordination may be necessary or desirable for present and future applications.

For instance, users in a conferencing system may sometimes decide to enter smaller discussion groups that may hold multiple, concurrent conversations [SFB⁺87]. A "brainstorming" session based on the "chalkboard" metaphor [SFB⁺87] in a cooperative environment is another example where less stringent coordination is appropriate. In the case of a multimedia-based group editor, strict coordination is not necessary when a user is browsing through a part of text while another is annotating a different part of the text. Thus, the degree of coordination needed varies from time to time within an application and across different applications.

3.1.1 Causal Synchronization

Both intra-flow and inter-flow coordination also require causal synchronization when more than two communicating entities are involved. For instance, if sender S_2 sends a message over a connection C_1 after "hearing" from sender S_1 over (say) another connection C_2 , all other participants must "see" messages from S_1 and S_2 in the proper causal order.

Enforcing causality requires that the underlying communication system buffer out-of-order messages and enforce the same total ordering on all related messages at all the sites. However, traffic delivery over multimedia connections (such as voice or video) usually has delay constraints and the messages not delivered within their delay bounds are rendered useless. Thus, causally ordered delivery mechanism must also take into account the real-time delivery requirements of multimedia traffic.

4 Our Approach

MCP (Multi-Flow Conversation Protocol) is a protocol suite consisting of transport and session layer protocols to facilitate multimedia communication.

MCP provides two methods of coordination: token-based control and an abstraction called *multi-flow conversation*. Together, these two methods yield a flexible and adaptable coordination mechanism.

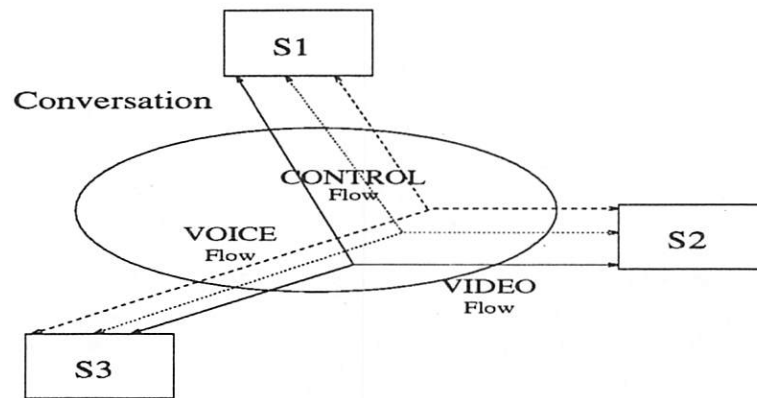


Figure 1: MCP conversation

4.1 Token-based Concurrency Control

When a multipoint flow is created, a token is assigned to that flow that acts as an authorization for data transmission. A sender must hold a token to be able to send traffic over a flow. However, token management primitives are provided so that other participants can obtain transmission privileges. Thus, applications are free to transfer, replicate, and delete tokens to govern the degree of concurrency control needed.

This type of concurrency control is entirely different from the token-based synchronization provided by OSI session-layer protocol. The latter method of synchronization allows session participants to insert resynchronization points (or checkpoints) in the data stream to allow rollback and to reduce the amount of retransmitted data in case of a transmission error.

We use the token mechanism to allow several levels of concurrency control. The following control hierarchy is derived based on schemes proposed in the literature [SFB⁺87, EGR89, Lan86, Swi88].

Floor Control As described earlier, real-time teleconferencing systems employ such a strict concurrency control. A single token enforces such control over a multipoint connection. The token will be passed on from one speaker to another whenever the floor is transferred. To obtain control of the floor, a participant must explicitly request transfer of the token by invoking a token management primitive.

Brainstorming This form of coordination is common in shared window systems where there is no concurrency control for simultaneous actions by multiple users. For such applications, the token is replicated and distributed to all the participants.

Chalkboard Interaction Applications based on the "chalkboard" metaphor in a cooperative environment interact in two phases. In the first phase, a speaker addresses a group of listeners with no interruptions. The application may switch to the second phase any time. In the second phase, a group of questioners may address questions to the speaker.

The token-based method can accommodate both types of interaction. Only the speaker may hold the token during the first phase, whereas the token may be replicated and distributed to the questioners during the second phase. Replicated tokens will be destroyed at the end of the second phase.

Discussion Groups Some environments such as real-time conferencing systems envisage a session breaking up into smaller discussion groups and thus holding multiple, concurrent conversations. In such a system, initially only a single token would be created and passed on from one speaker to another to achieve strict "floor-based" control. However, the token can be replicated and transferred to each discussion group, and each group may then use the token independently as it sees fit.

4.2 Multi-flow Conversations

To allow temporal synchronization among traffic over multiple flows, we introduce a communication abstraction called a *multi-flow conversation*. A conversation is a logical entity and consists of one or more

(two-party or multipoint) flows. An application will typically first create individual flows with appropriate performance requirements [FV90]. It will then create a conversation that includes one or more related flows to achieve the necessary temporal synchronization.

A conversation (Figure 1) may consist of one or more (two-party or multipoint) connections, MCP enforces temporal synchronization in delivery of traffic over participant connections. For instance, consider a conversation consisting of a multipoint voice connection V_1 and a data (text) connection T_2 . If a sender S , sends a message on T_2 followed by a message over V_1 ; all the conversation participants should receive those messages in the same sequence. In addition, a conversation provides limited causal ordering among messages sent by multiple sources. For instance, if S_2 sends a message M_2 over a constituent connection after "hearing" M_1 from S_1 , all other participants should "see" messages from S_1 and S_2 in the proper causal order.

However, enforcement of such a causal order among messages sent over a conversation has performance penalties (including delaying message delivery for a long time) and such penalties are not acceptable to real-time flows involving voice or video that have *better-never-than-late* semantics. For instance, real-time voice flows must deliver traffic within 100 milliseconds before quality of service degrades and traffic delivered after about 300 milliseconds is useless for voice interaction.

In fact, delivery of certain messages might be meaningless if delayed beyond the delay constraint of the flow involved. Therefore, we have decided to discard strict causal ordering in favor of a notion of causality called Δ -causality that takes into account the delay constraints associated with constituent flows.

Additional features of MCP include an out-of-band signaling channel for exchanging control information and an asynchronous interface to applications. Out-of-band signaling facilitates real-time protocol processing and simplifies processing of user data. The asynchronous interface allows an application to specify event-triggered actions to facilitate fast processing of periodic data and changes to the status of multipoint connections.

4.3 Delta-Causality

Informally, Δ -causality works as follows. Given a set of flows and a set of participants in a conversation, message delivery to an application at a recipient site is causally ordered provided an upper bound Δ on end-to-end delay is not violated for any of the related messages. For instance, in the example described above, the receiver S_3 will see the proper causal order provided M_1 and M_2 were generated and sent in such a way that their arrival at S_3 is not delayed beyond an upper bound Δ . Δ is a function of individual delay constraints for flows $Flow_1$ and $Flow_2$.

We assume that each flow has performance characteristics associated with it that are specified when a flow is established. Apart from the packet rate and error rate parameters, there are two delay constraints associated with each flow F_i :

Desired delay δ_i is the maximum end-to-end delay that the flow F_i can tolerate before quality of service deteriorates. Example of such a constraint is 100 milliseconds delay bound in packet voice.

Loss delay λ_i is an upper limit on end-to-end delay for flow traffic beyond which the delivered traffic delivery is useless. For example, packet voice or video have *better-never-than-late* delivery semantics and specify such a constraint. Packet voice traffic with desired delay constraint of 100 ms has a loss delay constraint in the range of 200 to 300 ms.

Consider a conversation C consisting of flows $F_1 \dots F_i \dots F_n$ with respective delay constraints δ_i 's and λ_i 's. We compute two conversation-wide desired delay and loss delay constraints:

$$\Delta_1 = \max\{\delta_i, i = 1, \dots, n\} \quad \Delta_2 = \min\{\lambda_i, i = 1, \dots, n\}$$

Based on these two constraints, we define the causality interval Δ for each conversation. Δ is computed as $\Delta_1 < \Delta < \Delta_2$ and defines a window of causality for each message sent in a conversation.

It must be noted that our notion of real-time assumes large-grain clock synchronization among the participants. This is not an unrealistic assumption as fault-tolerant clock synchronization algorithms exist

that achieve such synchronization. In TCP/IP Internet, Network Time Protocol (NTP) achieves global clock synchronization across the country within a few milliseconds [Mil89].

The value of Δ is chosen based on the following considerations. First, Δ must at least be equal to $\Delta_1 + 2 * \epsilon$ (ϵ is the maximum possible clock skew). Second, to allow some flexibility in the presence of fluctuations in network conditions and delays, MCP provider may choose value of Δ to be higher than $\Delta_1 + 2 * \epsilon$ without compromising individual flow semantics as long as Δ remains much below the upper bound Δ_2 .

4.4 Design Issues

Given the need for flexible coordination and synchronization mechanisms, we carefully considered the following design alternatives:

Policy vs. Mechanism: One of the questions we had to address was whether to implement the necessary policies in the communication substrate or to build only the necessary communication mechanisms that are flexible enough to allow an application to choose and enforce desired policies.

We believe that the right way for accommodating a wide range of coordination and synchronization requirements is to provide communication mechanisms that allow specification and selection of the degree of synchronization needed and leave the task of choosing the appropriate policy and degree of synchronization to each individual application. Such a division of functionality is appropriate because an application programmer has the relevant knowledge to specify the necessary conditions for causality.

Kernel vs. Library: Given the availability and popularity of existing transport protocols such as TCP or UDP, we also had to grapple with the issue of whether we could implement MCP functionality in an application layer library on top of conventional transport protocols. Advantage of such an approach is two fold. First, no modifications are necessary to the kernel allowing easy portability to different systems. Second, a library-level implementation can always hide unpleasant details of coordination and synchronization from an application and thus let application writer concentrate on application-specific details. Library level implementation is also easier to modify allowing experimentation with alternative designs.

However, library-level implementation approach suffers from some drawbacks. A distributed multimedia application must handle and coordinate transmission and reception of traffic over multiple data streams and must have the ability to handle asynchronous events that occur on any one of the streams. If you consider details of rate-based flow control, buffer and timer management, and asynchronous token management, even a multi-threaded library implementation becomes quite unwieldy. Some degree of inefficiency also results as the library must duplicate some of the kernel functions such as event scheduling and buffer/timer management.

Based on these arguments and our previous experience with a library-based implementation of an experimental transport protocol [YD91], we decided in favor of the kernel-based implementation.

5 MCP Implementation

One of the goals of our research is to integrate MCP abstractions into a general purpose operating system such as Unix so that we can utilize a large collection of toolkits and application software libraries [Dew90] to build interactive multimedia applications. Towards that goal, we have implemented MCP in the framework of Unix 4.3 BSD networking software.

5.1 Implementation Issues

Unlike the implementation of TCP and UDP protocols in a Unix environment, the MCP protocol makes some additional demands on networking software:

1. MCP must support both point-to-point (or *two-party*) and multipoint (or *N-party*) flows. In the case of multipoint flows, a flow is identified by a group address and the protocol software must be

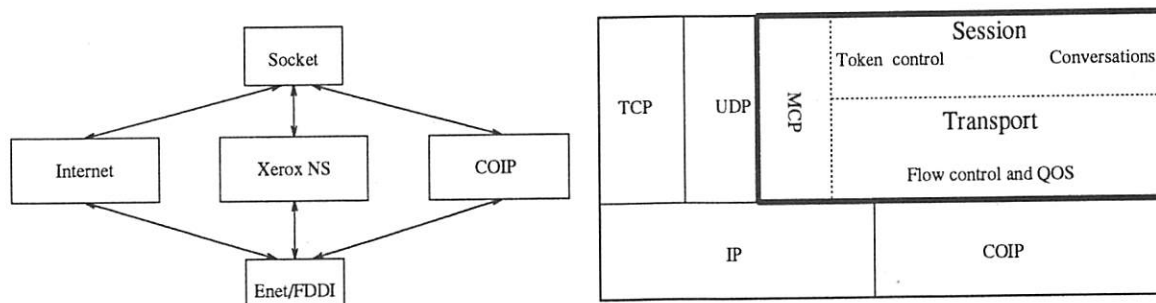


Figure 2: Organization of Unix networking software layers is shown on left hand side. Figure on right hand side shows the position of MCP in DARPA Internet hierarchy.

careful in multiplexing and demultiplexing of packets addressed to the group address. For instance, if three transport level entities at a host may participate in the multipoint flow, each endpoint must be separately identified and a copy of an incoming packet must be delivered to each local participant. MCP must also clearly define the loopback semantics for an outgoing packet over such a flow. For instance, the loopback may mean that all the local participants other than the sender of the packet receive a copy of each outgoing packet.

2. MCP application-layer interface must make provisions to allow specification of QOS parameters such as data rates, error tolerance, and delay bounds for individual flows and to specify control operations such as token request/transfer and joining or leaving an ongoing conversation.

Given the QOS specifications, MCP must exercise rate-based flow control according to the QOS specification and include appropriate timer and buffer management functions to implement Δ -causality.

3. MCP must also make provisions for reporting asynchronous events and exceptional conditions to user applications. For example, arrival of a token or token transfer request can occur any time. Because MCP assumes that a user application implements the desired policy for token management, MCP must asynchronously report such events to the application.
4. The size of data units transmitted varies depending on the media used. For example, a voice sample may consist of a few bytes whereas a video frame typically consists of hundreds of bytes. To ensure proper temporal synchronization, MCP must preserve message boundaries over individual flows and insert synchronization markers to ensure appropriate playout of related media units at a receiver.
5. Because the transmission and delivery of data over a multipoint flow is controlled by transfer of tokens and relevant control information, processing of normal data is complicated if one uses "in-band signaling" as in traditional transport protocols. To facilitate real-time protocol processing and to simplify processing of user data, MCP must establish an auxiliary control connection for communicating out-of-band control messages related to conversation and token management.

5.2 Unix Networking Software

Networking in the BSD kernel is divided into three software layers (Figure 2): The socket layer, the protocol layer, and the network layer. A socket acts as an endpoint of communication and is accessible to processes as a Unix file descriptor. The socket layer is usually invoked by the C library using a set of system calls. The protocol layer consists of protocol suits or domains such as TCP/IP, Xerox NS, and Decnet domains. This layer handles all protocol specific processing. The network interface layer manages the physical network such as ethernet or token ring. Figure 2 shows the MCP's place vis-a-vis other protocol modules and additional details on Unix network software design can be found in [LMKQ89].

In addition to the standard Unix software, two extensions that support network layer connections (ST-II and COIP-K) are available in public domain [PP92, Cra92]. Because we want to provide delay-constrained delivery of multimedia data, we needed a network layer service that would reserve resources to guarantee

CLIENT	SERVER
<code>s = socket(PF_COIP, SOCK_MCP, 0)</code>	<code>s = socket(PF_COIP, SOCK_MCP, 0)</code>
<code>s = setsockopt(s, MCP, MCP_SETFLSPEC,</code> <code>&flowspec, sizeof(flowspec))</code>	<code>.....</code>
	<code>bind(s, addr, addrlen);</code>
<code>connect(s, addr, addrlen);</code>	<code>listen(s,5);</code>
<code>/* Establish a conversatioon */</code>	<code>snew = accept(s, addr, addrlen);</code>
<code>setsockopt(s, MCP, MCP_INITCONP, sockarray, sizeof(sockarray));</code>	

Figure 3: A typical multipoint MCP application consists of a single client and one or more servers. Servers are passive listeners waiting for the client to establish a connection. Once a connection is established, the token holder gets to transmit and a server (or the client) must first obtain a token before sending any data.

performance. From that point of view, both ST-II and COIP provide necessary service. We decided to use COIP protocol because COIP protocol implementation consists of a toolkit called COIP-K that provides a nice modular structure and a flexible platform to implement connection oriented protocols.

5.3 Socket Library and Programming Interface

MCP retains the standard 4.3 BSD socket interface as much as possible. We have extended the socket library to provide additional functionality needed by MCP applications. In particular, we have extended `connect` call to allow specification of multiple destination addresses for a multipoint connection. We have added new options to the `setsockopt` call so that an application can specify a flow's QOS requirements. `setsockopt` call is also used to request a token, to explicitly transfer a token to another participant, or to join/leave a conversation.

We have also extended `getsockopt` to implement an asynchronous event notification mechanism as described later in Section 5.5.4.

MCP retains the conventional TCP/IP programming interface in Unix. Thus, applications involving multipoint communication are also written using client-server paradigm. In a multipoint application, a single participant acts as a client and all other participants act as servers (or passive entities). An example of such a program is shown in Figure 3. Each server uses a combination of `socket`, `bind`, `listen`, and `accept` calls to indicate its willingness to join a multipoint flow.

After one or more flows are established, participants may add the flows to a conversation. Once a conversation is established, the data transfer begins using the usual Unix `send/recv` calls. Before sending data, A participant must typically request a token using the `setsockopt` call and a token holder may transfer the token using another `setsockopt` call (see Figure 5).

5.4 MCP Transport and Session Layer Implementation

MCP transport and session layers reside below the socket layer. The following discussion on transport and session layer implementation assumes that readers are familiar with basic structure of Unix network software including the two commonly used data structures, *Protocol Control Block* (or PCB) and *mbuf* (Memory Buffer). PCB is used to hold protocol-specific state information and *mbuf* is another useful data structure used to shepherd an incoming or outgoing packet through protocol layers without repeated copying. More details can be found in [LMKQ89, Cra92].

Internal structure of MCP implementation is shown in Figure 4. For simplicity, only selected functions are shown. The function `mcp_userreq` handles all user requests such as socket creation, bind, connect, and read/write and passes on these requests to lower layer functions. For instance, `cin_output` at network layer transmits data to other endpoints of a flow. Other functions implement token management and conversation abstraction as described below.

On input side, the routine `cinintr` handles packet arrivals at the network interface and passes data to the transport layer. The routine `mcp_ctxext` enforces the causality requirement. The routine `mcp_ctlinp` handles data transfer across the control connection.

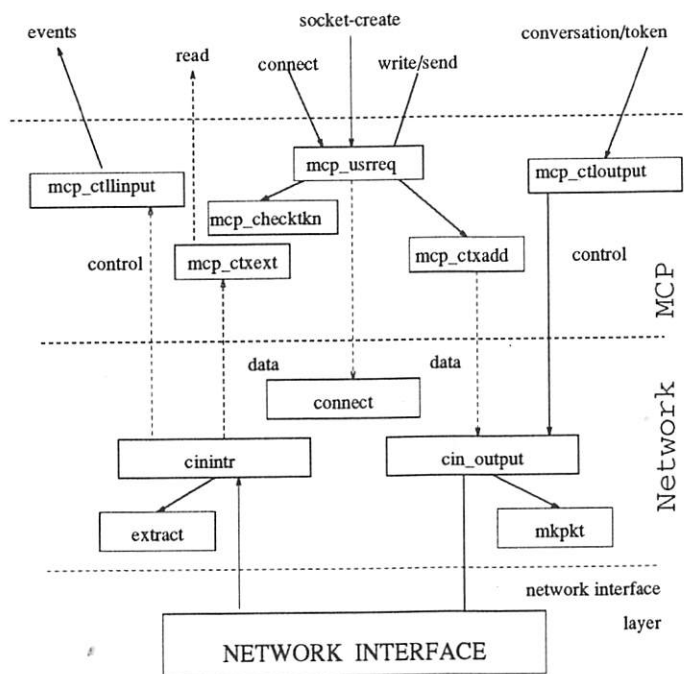


Figure 4: Internal structure of MCP implementation

5.4.1 Transport Layer Implementation

MCP transport layer is responsible for establishment of point-to-point and multipoint data and control flows and for the transfer of data to and from the MCP session layer. MCP transport layer identifies each endpoint of a flow using a unique transport level address consisting of $\langle ip_addr, port_no \rangle$.

When a client issues the **socket** call followed by a **connect** call, MCP transport layer traces the following steps to establish a multipoint flow¹:

- The transport layer creates a MCP PCB (see Figure 5) to maintain the state information about the flow and fills in flow specifications, list of flow participants, and initial status of token.
- MCP transport layer assigns the flow a unique flow identifier. A unique flow identifier is constructed by appending a local sequence number to the IP address of the client's site.
- The transport layer constructs an MCP open packet, fills in flow identifier and flow specifications, and invokes the COIP-K network layer to establish a multipoint flow passing the open packet as an argument. A multi-point flow is implemented in one of two ways.

Under the first approach, COIP-K builds a tree with client (active entity) as a root that connects the group participants using point-to-point connections. Every participant forwards a message addressed to the group to the entity at the root and the network-level entity at the root delivers a copy of the message to the local participants and also forwards copies of the message to other leaves across a separate point-to-point connection.

Under the second approach, MCP uses IP multicast [Dee88] to send and receive packets over the flow. Thus, in this approach, there is no master-slave relationship. However, additional network layer functionality [ZDE⁺93] is necessary to provide performance guarantees. Given a destination group address, the IP multicast routing module (added to COIP-k) ensures that a packet reaches all the members of the multicast group. Current implementation does not include error control to ensure reliable multicast delivery and is the focus of our current research.

¹Establishment of a point-to-point or two-party flow is similar except packet delivery is straightforward as in TCP or UDP. Separate discussion of point-to-point flows is omitted here.

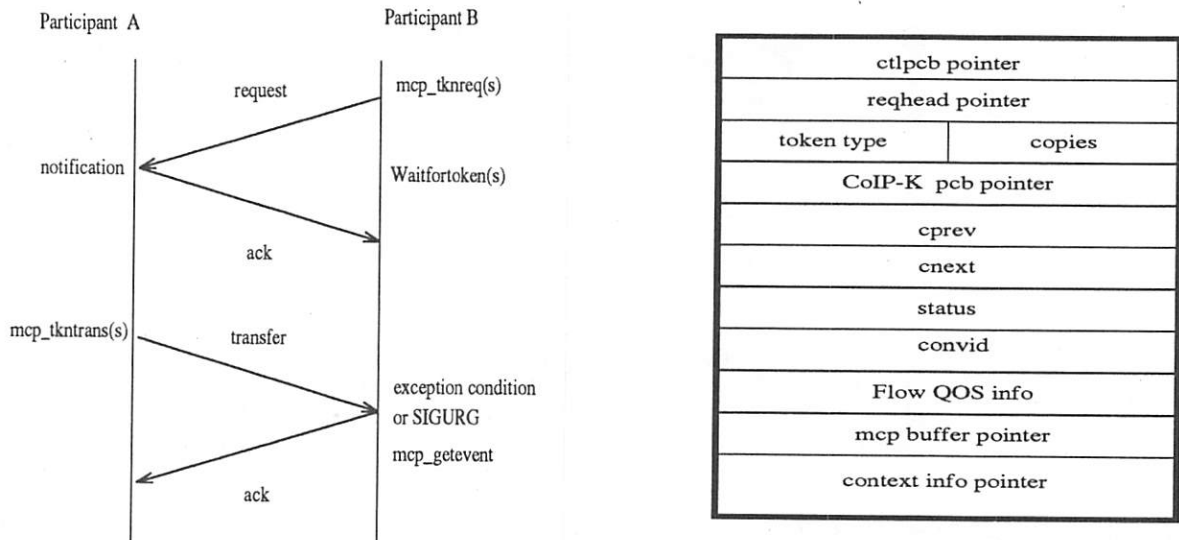


Figure 5: Example of token request and transfer using setsockopt call and unix signal is shown above. MCP PCB (Protocol Control Block) is shown on right hand side.

- Given a conversation, MCP also establishes a separate control flow among the transport-layer peers at participating sites. The control flow is devoted to exchange of state and control information related to the conversation and typically has different packet delivery semantics. Communication over a control flow is typically only two-party and requires reliable delivery. When a conversation is created, MCP transport layer uses the COIP-K layer to establish a control flow and the open packet sent to participating sites carries a list of flow identifiers for the constituent flows of the conversation.

5.5 MCP Session Layer Implementation

MCP session layer implements two important functions, namely, token management across a multipoint flow and temporal/causal synchronization among messages sent across constituent flows in a conversation.

Data transmission across a flow at each sender consists of a sequence of messages. Data sent using each `write` call by an application constitutes a single message. The MCP session layer preserves the message boundaries in delivery of data at each participant and also uses the message boundaries as synchronization markers (or points) for both temporal and causal synchronization.

5.5.1 Temporal Synchronization

Given a flow, MCP session layer at each sender assigns a unique sequence number to each message sent across the flow. When multiple flows are part of the same conversation, sequence numbers for messages sent by the same sender across constituent flows are drawn from a single sequence number space. Thus, MCP session layer can deduce the message ordering among messages sent by the same sender across constituent flows of a conversation.

When a sender S_1 sends two messages M_1 and M_2 in that order across two constituent flows F_1 and F_2 , MCP implements temporal synchronization by delivering them in the same order at all the participants. Because message sequence numbers capture the necessary temporal order, implementation of temporal synchronization is straightforward. However, if a preceding message M_1 is lost or arrives out-of-sequence, MCP session layer at the destination must buffer the subsequent message M_2 and decide when to deliver it to the application. Because each flow QOS specification specifies the delay bounds for individual messages and the packet rate, MCP uses the information to derive a delivery deadline for each message. Thus, M_2 is buffered only until its deadline and is delivered to the application (with an exception reported) if M_1 fails to arrive by the deadline.

5.5.2 Token Management

For each participant of a flow, MCP allocates a separate MCP PCB that maintains the state information about the token for the participant. MCP enforces token semantics in the following way:

- When a sender tries to send a message across a flow using MCP, the MCP session layer transmits the message only if the sender PCB holds a token. Otherwise, MCP returns an error indicating that the token is not present and it is left to the application to explicitly request transfer of a token from the current token holder.
- When a token request is made, the MCP session layer broadcasts the request across the multipoint flow. When the request is received by MCP layer at a token holder, it immediately acknowledges receipt of a request so that the sender may not repeatedly broadcast its request. MCP layer at the token holder then notifies its application using an asynchronous event notification mechanism that identifies the requesting participant.
- MCP does not implement any particular token management policy and it is left to the application to decide whether and when to transfer the token to the requesting entity.
- Token replication requests are handled locally by updating appropriate fields in the MCP PCB for the participant. MCP also implements other token management calls such as `copy_token`, `delete_token`, and `distribute_token`.

5.5.3 Δ -Causal Synchronization

When multiple senders are involved in a conversation, MCP session layer implements a limited notion of causality called Δ -causality.

Whenever a conversation participant *A* sends a message M_1 , MCP session layer at *A* includes some *context* information in the message header. The context information included with each message is a pair `<sender_id, msg_id>` for each participant (or sender) in the conversation. That is, the context includes the sequence number of the last message received at *A* from each participant before sending M_1 (see Figure 6).

Thus, when M_1 arrives at another participant *B*, the MCP session layer does the following:

1. MCP at *B* checks to see whether or not it has already received all the messages specified in the context of M_1 . If yes, MCP schedules M_1 for delivery.
2. If not, MCP can build the dependency list of messages for M_1 . Figure 6 shows an example.
3. Pending reception of previous messages in the context, MCP computes the delivery deadline of M_1 using the Δ value, sets a timer to go off at the deadline, and buffers M_1 . M_1 is buffered in a message dependency list according to its position with respect to pending messages.
4. Whenever another message M_2 arrives over the conversation, MCP checks the head of the dependency list to see whether there are any messages waiting for the arrival of M_2 . If so, those packets are dequeued and are scheduled for delivery along with M_2 according to their delivery deadlines. If message M_2 itself contains some unresolved dependencies, it is also queued up (see Figure 7).
5. When a timer goes off for a message M , MCP dequeues M and any messages that were queued behind M according to dependency information and queues them up for delivery to the application layer.

5.5.4 Asynchronous Event Notification

One of the interesting MCP implementation issues concerned an user interface concern. A multimedia application involving multiple data streams must simultaneously handle several events including arrival of traffic over individual flows, mouse and keyboard events, and asynchronous events such as token arrival or a token transfer request. Conventional approach to building such applications in Unix is to use the `select` system call which is clumsy at best.

Message from A has following context info

Flow id	Sequence number	Participant number	B's Info	C's Info	D's Info
5	18	A	10	2	12

Context info at B

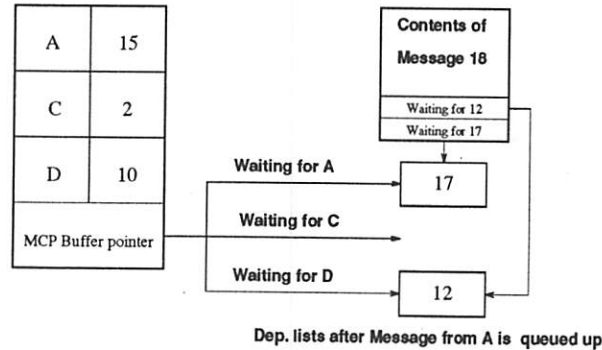


Figure 6: At the top, figure shows the context information included by A in its message number 18. Context information in the message specifies that the message should be delivered ONLY after message number 17 (from A), message number 2 (from C), and message number 12 (from D) have been delivered. However, the context information at B indicates that B has received so far first 15 messages from A, first 2 messages from C and first 10 messages from D. Therefore, message 18 from A is queued and added to the dependency lists.

It is our belief that system designers must allow an upcall-based user interface to allow notification of asynchronous, event-triggered actions and MCP assumes such an interface. Under such an interface, an MCP application can specify an action to be taken when a particular event occurs. For example, the MCP service primitive `flow_status(<status change type>, <procname>)` specifies the upcall procedure to be called when one of the pre-defined status changes or events occurs on a flow. The arguments to the `procname` include flow identifier, an integer code specifying the kind of change, and optionally the user data received. However, Unix does not support upcalls. We are currently investigating alternate ways of implementing upcalls in Unix. Currently, such a facility is implemented in a MCP library routine. The library routine in conjunction with MCP kernel simulates an upcall as follows. MCP reports an asynchronous event by sending a signal (`SIG_URG`) to the user process and setting the `outofband` flag in the socket structure. The `upcall_handler` library routine first uses `getsockopt` call to retrieve the cause of the event and any additional information from the kernel, and then calls the corresponding upcall routine that is previously registered by the application.

6 Experience with MCP

The experimental set up for MCP prototype consists of four SUN Sparc ELC workstations equipped with audio devices on an ethernet in our Distributed Systems Laboratory. The laboratory is mainly used for experimental research and consists of a dozen HP and Sun workstations with three Network file servers.

We have implemented many multi-party, interactive applications to test/debug MCP kernel, to evaluate the effectiveness of MCP abstractions, and to measure performance of multimedia applications with and without using MCP. In the following, we describe our experience using MCP and Section 7 discusses the results of our performance evaluation.

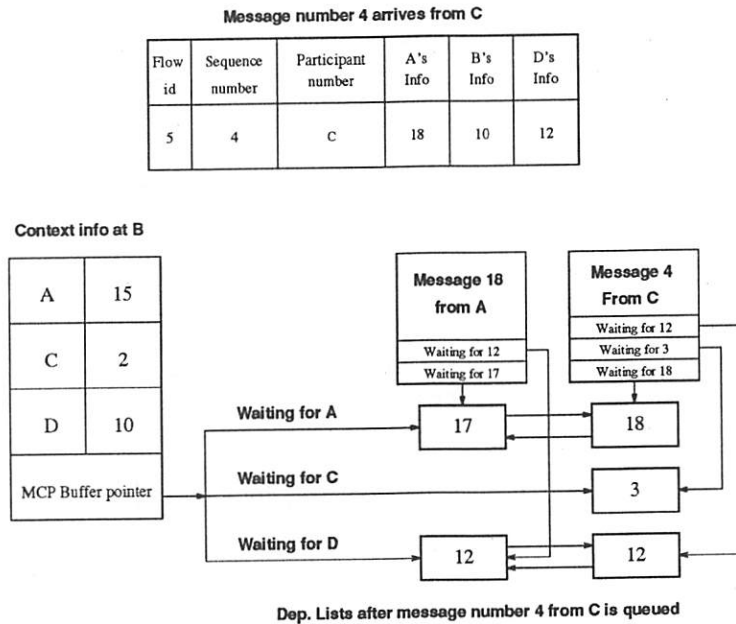


Figure 7: Figure shows MCP actions when another message (seq. no. 4) arrives from C. The new message depends on the messages 18 and 12 from A and D respectively.

6.1 Simple Tests

To help us test and debug various parts of MCP implementation, we wrote two simple applications, namely, *mshell* and *mtalk*. *Mshell* is an interactive shell that provides command-level access to various MCP primitives such as flow creation, conversation creation, joining/leaving a conversation, and token request/transfer. Thus, *Mshell* was useful in testing each MCP service primitive separately and in various combinations with other primitives.

Mtalk (for Multi-user talk) is a multi-party conferencing program that allows a conversation among multiple users [Cra92]. At each user, the program divides the screen into two parts: an input region and display region. Lines of text typed in input region are transmitted over a MCP flow to all the other users and text from other users is displayed in the display region of the screen. In particular, we used *mtalk* to test our implementation of multi-point flows using IP multicast.

6.2 Nevot

NEVOT (Network VOice Terminal) is a network voice terminal that supports multiple concurrent conferences (both two-party and multi-party) on top of a variety of transport protocols and using a range of audio encodings from vocoder to multi-channel CD quality. It is developed as an experimental tool by Henning Schulzrinne of Univ. of Massachusetts and has been implemented on top of TCP, UDP and ST-II.

We have ported Nevot to run on top of MCP and used Nevot/MCP in two ways. First, Nevot allowed us to gain some experience in using MCP token management primitives in implementing different conference control styles. Second, we evaluated the cost of using MCP by comparing the performance of audio conferencing applications implemented on top of Nevot/MCP, Nevot/UDP, and Nevot/MCP.

We built an audio conferencing application and experimented with the following conference control styles:

Floor Control: This is a user driven conference control style used commonly in teleconferencing applications. Under this model at most one speaker holds "floor" at any time and is allowed to talk at any time. Other users must request control of floor before speaking. MCP supports such a user-driven control using tokens. For each flow, there is only one token allocated initially and the applications can transfer token from one speaker to another based on a policy that is chosen by the application. In addition,

MCP allows an application to make multiple copies of a token and distribute copies among few participants. In Nevot, for instance, we can implement a hierarchical conference control where a subgroup of participants have access to the floor at any time and can separately control arbitration of speaking rights among the members of the subgroup.

Our overall experience so far has been that token management in MCP works very well in implementing a range of floor control policies. However, users must be made aware of token management because a speaker must typically be told to wait until a token is requested and obtained from another speaker.

Activity Sensing: Floor-control based conference control using explicit transfer of tokens is not suitable for many collaborative multimedia applications involving shared windows, image browsers, and group text editors. In such an environment, strict control is desired to ensure that at most one user is allowed to manipulate parts of a shared workspace. However, transparent transfer of transmission rights is desirable whenever a user finishes her update (idle for some time), or current sender of data moves mouse (focus) away from a shared window.

An example of transparent floor control is a *distributed, voice activated, collision sensing algorithm* proposed in [CLS⁺93]. Under an activity sensing algorithm, the underlying software transfers token to another requestor when it notices that the token holder has been idle for some time. However, such an algorithm requires choosing a correct time interval to decide whether or not a speaker is in idle (quiescent) state. Our experience showed that an idle interval of 200 to 400 milliseconds gives good results in terms of detecting quiescence.

The algorithm works well only in a local area environment and may not work so well when propagation delays are large over a wide area network (WAN). We are now trying out several variations on activity sensing algorithms.

Brain Storming: An extreme alternative to strict floor-based control is *free for all* in which no explicit conference control is exercised and all participants are free to transmit whenever they wish. We have found this to be a useful and efficient form of conferencing especially when a conference enters a discussion phase where there is no obvious speaker at any time.

6.3 Tests involving Δ -causality

To test Δ -causality, we wrote another application that created a MCP conversation consisting of three multipoint flows: a background flow, F_{file} that repeatedly transferred a large file, a background audio flow F_{au} that continuously played back file containing pre-recorded music, and an interactive $mtalk$ flow in the foreground involving at least three participants. The tests involved simulating variable delays and random packet loss inside the kernel. In the experiments, about 1% packets were chosen at random for drop and, in addition, the end-to-end delay was varied between 20 milliseconds to 500 milliseconds. The parameter Δ was chosen to be 200 milliseconds which happens to be an acceptable delay for packet voice before delays cause noticeable breaks in audio reception.

The tests served two purposes. First, they helped us verify correctness of our implementation of causal and temporal synchronization. For instance, MCP kernel estimates the maximum buffering necessary to buffer packets delayed until their Δ deadlines and also computes the rate of packet delivery based on QOS specifications. Tests allowed us to verify that buffer estimates were sufficient and packet rates could be maintained in a non real-time operating system such as Unix. Second, we wanted to see how does enforcement of Δ -causality affect the audio playback at participants in the presence of background file transfer and bursty interactive traffic. Our experience is that MCP can easily maintain the deadlines associated with Δ -causality in the presence of occasional packet loss. Packets are lost either due to random drop (network errors) or because packets that arrive beyond the delivery deadline are discarded. Assuming resource reservation at the network layer, we expect the packet loss and delays in future high-speed Internet to be of similar nature. Tests also showed that occasional loss of audio packets did not cause easily perceptible degradation in playback quality. However, more systematic and quantitative measures are necessary to evaluate impact of such losses.

	TCP	UDP	COIP-RAW	MCP-kernel
Loopback (ms)	1.68	1.28	1.26	.96
Ether (ms)	1.80	1.47	1.36	1.07

Table 1: The columns show the cost of transferring a token from one participant to another located across the ethernet or on the same machine (loopback). The values represent an average computed across 100 token transfers and experiment was repeated several times to verify that the results are consistent.

TCP	UDP	COIP-RAW	MCP
1.9	1.9	1.9	46

Table 2: The columns show the cost of replicating a token in microseconds. Replicating a token only involves updating a field in MCP PCB and no communication. Cost of token replication is higher with MCP kernel implementation because it involves a system call as compared to simply updating a variable in MCP library in case of first three implementations.

7 Performance Evaluation

Goals of our performance evaluation were two-fold. First, we wanted to determine the impact of using MCP on a multimedia application. Specifically, we were interested in comparing the performance of an application implemented using MCP against the performance of the same application implemented using other transport protocols such as UDP and TCP. Second, we wanted to compare the performance of MCP kernel implementation against implementation of MCP as a run-time library.

In the following, we describe the results of our performance evaluation. Whenever end-to-end measured values for an operation were too small for clock resolution, we repeated the operation several times and computed the average value.

7.1 Comparison with Other Protocols

We have implemented MCP services as part of a run-time library on top of three different protocols: TCP, UDP, and COIP-RAW. COIP-RAW provides direct access to COIP network interface through Unix sockets. We ported Nevot to run on each of the MCP library implementations and on top of MCP kernel.

Tables 1 through 4 show the results involving token operations and use of Nevot. Tables 3 and 4 show results of a Nevot experiment involving a sender and a receiver. Sender transmitted contents of an audio file (audio playback) in a loop. Nevot prints statistics collected using Unix `getrusage` call after every 20000 packets and statistics include application's user and system times. Packets arriving late at the receiver are discarded. MCP kernel performance is comparable to that using other protocols. On the receiving side, Nevot/TCP loses packets due to late arrivals even across an ethernet because TCP queues up data at sending side to send in chunks. Using `TCP_NODELAY` option prevents TCP from queuing up data and that reduces the number of late packet arrivals, but the total times are then much worse as shown in Tables.

7.2 Kernel vs. Library Implementation

To evaluate the impact of implementing MCP in a kernel, we also implemented MCP kernel as a run-time library on top of COIP-RAW interface. A multimedia application using MCP library is implemented as

	TCP	TCP (NO_DELAY)	UDP	COIP-RAW	MCP-kernel
user time (sec)	45.3	45	41.8	35	35/45
system (sec)	29.6	41	33.6	34.3	35/43

Table 3: Statistics at the Nevot sender side for 20000 packets sent at the rate of 50 packets per second. Each packet contains an audio sample worth 164 bytes. The second (larger) time for nevot on the MCP kernel take into account the cost of activity sensing.

	TCP	TCP (NO_DELAY)	UDP	COIP-RAW	MCP-kernel
user time (sec)	38.2	38.5	37.9	38.7	38.8
system (sec)	29.2	40.8	29.9	33	33.35
late packets	21-11	3	3	5	5

Table 4: Statistics at the Nevot receiver side. At the receiver side, row labeled *late packets* shows the number of packets that arrive too late for playout.

Type of Connection	Implementation level	
	MCP Library	Kernel
Point to Point	35.6	6.7
Multipoint	57.6	4.5

Table 5: *The Conversation create time (in milliseconds) shows the amount of time needed to establish a single-flow conversation involving two participants in each case. Establishing a conversation consists of creating a control flow and updating MCP state to add the flow to a conversation. Typically, a conversation consists of more than one flow, but the cost of creating a multi-flow conversation is similar to that of creating a single-flow conversation.*

a multi-process application where one thread is responsible for handling data transfer across a flow in a conversation. MCP state common among threads is maintained using shared memory.

Thus, creating a conversation requires starting a new process to manage the control flow and communication with remote participants to establish a conversation. Implementing library routines through multiple threads of an application simplifies programming and makes it easy to hide the details of handling data transfer and asynchronous events over individual flows. However, as Tables 5 and 6 shows, cost of operations such as creating a conversation and data transfer is significantly higher using the library as compared to the kernel implementation.

8 Related Work

The work described in this paper is related to current research in two main areas: user interface for coordination and conference control and OS and protocol support for enforcement of QOS and temporal/causal synchronization.

In the area of conference control and collaboration, CECED [CLS⁺93] system is a notable example of a software system that supports collaboration involving multiple users with minimal intrusion of software and user styles. For coordination and floor control, CECED uses an activity sensing algorithm as described earlier. We have used the algorithm as a basis for further experimentation with alternate activity sensing policies.

Schulzterrinne [Sch92] discusses several issues involved in designing a real-time transport protocol (RTP) to support audio conferencing across the Internet. RTP's main focus on achieving playout synchronization. MCP design [Yav92] precedes RTP design and differs from RTP because MCP considers causal synchronization in addition to the temporal synchronization.

Recent work at Tenet group in Berkeley [SV93] has identified some fundamental requirements of mul-

Type of Connection	Implementation Method		
	MCP Library (COIP-RAW)	MCP Library (TCP)	Kernel
Point to Point	257	400	78
Multipoint	441	-	185

Table 6: *Data Transfer time for 100 packets each containing one byte of information. All the timings are in milliseconds.*

timedia collaborative applications including conference management, QOS mechanisms, support for NXM (N senders and M receivers) data flows, and so on. MCP provides functionality to meet several of those requirements and although MCP currently supports only symmetric NXN flows, it can easily be modified to support NXM flows.

MMCC, Multimedia Conference Control system, developed at USC/ISI [Sch93] concentrates only on providing session level conference management functions such as conference establishment, inviting parties to join a conversation, merging conferences and starting a "side-conference" through an application level connection manager. MCP includes most of such support and, in addition, provides communication mechanisms to enforce several varieties of conference control and to achieve synchronization among several flows.

A system developed by Angebrannt and others [AhLS91] shares some of the goals of MCP. Their system contains an audio protocol that supports audio connections and virtual devices. Synchronization in that system refers to synchronizing beginning of playout at several audio devices and achieving lip-syncing. MCP does not contain mechanisms to synchronize playback starts at several sites, but provides a general purpose temporal and causal synchronization facility irrespective of whether media sources are of fixed rate or totally asynchronous variety. Another nice aspect of Angebrannt's system is that their programming interface supports a set of command queues (or scripts) that allow specification of synchronization among audio devices. The system also reports several types of events including device operation completions, arrival of synchronization points, and so on. MCP, on the other hand, only considers a subset of communication events related to token passing and changes in membership of a flow or a conversation.

9 Summary

We have described design and implementation of MCP session and transport layer protocols that provide communication mechanisms and abstractions useful for building distributed multimedia applications. We have implemented MCP in SUN OS 4.1.1 kernel and evaluated its performance and suitability of its mechanisms for building multimedia applications.

Our work is significant in following respects. First, to the best of our knowledge, this is one of the first Unix-based implementation of transport and session level services designed explicitly to meet the coordination and temporal/causal synchronization needs of multimedia applications. Second, we have shown that coordination and synchronization mechanisms can be efficiently implemented in a general purpose OS such as Unix. Third, we have identified alternate conference control styles that should be supported to meet a wide range of applications including audio conferencing, group editors, and shared workspace. We plan to continue further research in various conference control styles and user interfaces for collaboration using multiple media.

During our work, we have also discovered some of the limitations of the process scheduling and event notification mechanisms in Unix-like operating systems. For instance, Unix lacks an upcall mechanism and it is not easy to promptly notify a process of an asynchronous event such as arrival of a token or a token request. We have identified some possible extensions to Unix scheduling mechanism to overcome such a problem and plan to pursue further research in that direction.

Acknowledgments

Authors would like to thank Charles Kranor and Guru Parulkar of Washington University in St. Louis for making COIP-K sources available to us and for cheerfully putting up with our questions about COIP-k software during the initial stages of MCP implementation.

References

- [AhLS91] Susan Angebrannt, Richard L. hyde, Daphne Huetu Luong, and Nagendra Siravara. Integrating audio and telephony in a distributed workstation environment. In *USENIX '91 Conference Proceedings*, pages 419-435. USENIX, June 1991.
- [CLS⁺93] E. Craighill, R. Lang, K. Skinner, M. Fong, and J.J. Garcia-Luna-Aceves. CECED: A System for Informal Multimedia Collaboration. unpublished report, SRI International, CA, January 1993.

- [Cra92] C. Cranor. An Implementation Model for Connection-Oriented Internet Protocols. Technical report, Department of Computer Science, Washington University, May 1992. MS Thesis.
- [Dee88] S. E. Deering. Multicast Routing in Internetworks and Extended LANs. In *Proceedings of ACM SIGCOMM '88 Symposium*, August 1988.
- [Dew90] Prasun Dewan. A Guide to Suite: Version 1.0. Technical Report SERC-TR-60-P, Software Engineering Research Center, Purdue University, February 1990.
- [EGR89] C. Ellis, S. Gibbs, and G.L. Rein. Design and Use of a Group Editor. In *IFIP WG2.7 Working Conference on Engineering for Human Computer Interaction*. North Holland, August 1989.
- [FV90] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3), April 1990.
- [Lan86] Keith A. Lantz. An Experiment in Integrated Multimedia Conferencing. In *Proceedings of Conference on Computer-Supported Cooperative Work*, pages 267-275, December 1986.
- [LMKQ89] Samuel J Leffler, Marshall Kirk McKusick, Michael J Karels, and John S Quatarman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison Wesley, 1989.
- [Mil89] Dave L. Mills. Measured performance of the network time protocol in the internet system. Network Working Group Request for Comments: 1128, October 1989.
- [PP92] Craig Partridge and Stephen Pink. An Implementation of the Revised Internet Stream Protocol (st-2). *Journal of Internetworking Research and Experience*, March 1992.
- [Sch92] H. Schulzrinne. Issues in designing a transport protocol for audio and video conferences and other multiparticipant real-time applications. INTERNET-DRAFT, IETF Audio-Video Transport Working Group, December 1992.
- [Sch93] Eve M. Schooler. Case study: Multimedia conference control in a packet-switched teleconferencing system. *Journal of Internetworking: Research and Experience (1993)*, 1993.
- [SFB⁺87] Mark Stefik, Gregg Foster, Daniel Bobrow, Kenneth Kahn, Stuan Lanning, and Lucy Suchmann. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Communications of the ACM*, 30(1):32-47, January 1987.
- [SV93] Clemens Szyperski and Giorgio Ventre. A characterization of multi-party interactive multimedia applications. Technical report, The Tenet Group International Computer Science Institute and Computer Science Division UC Berkeley, February 1993.
- [Swi88] Daniel C. Swinehart. System Support Requirements for Multi-media Workstations. In *Proceedings of the SpeechTech '88 Conference*, pages 82-83, New York, April 1988. Media Dimensions, Inc.
- [Yav89] R.S. Yavatkar. *An Architecture for High-Speed Packet Switched Networks*. PhD thesis, Purdue University, August 1989. Also available as TR-898.
- [Yav92] Raj Yavatkar. MCP: A Protocol for Coordination and Temporal Synchronization in Multi-media Collaborative Applications. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE, June 1992.
- [YD91] R. Yavatkar and V. Dumasia. Reliagram - A Communication Abstraction for Distributed Processing. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, December 1991.
- [ZDE⁺93] L. Zhang, S. Deering, D. Estrin, S. Schenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. submitted to ACM SIGCOMM '93, March 1993.

K. Lakshman is a graduate student in the Department of Computer Science at the University of Kentucky, Lexington, KY. He received his B.E. degree in Instrumentation Technology from the University of Mysore in 1989. His research interests include Communication and OS support for multimedia systems.

Giri Kuthethoor is a graduate student in the Department of Computer Science at the University of Kentucky. He received his B.E. degree in Computer Science from the University of Mysore in 1990. His research interests include Communication Protocols and Multimedia Software.

Raj Yavatkar received a Ph.D degree in Computer Science from Purdue University in 1989. He is currently an Assistant Professor of Computer Science at the University of Kentucky where he directs research efforts in high-speed networks, distributed shared memory systems, and multimedia software Systems.

The Spring nucleus: A microkernel for objects

Graham Hamilton Panos Kougiouris

Sun Microsystems Laboratories, Inc.

Mountain View, CA 94043 USA

Abstract

The Spring system is a distributed operating system that supports a distributed, object-oriented application framework. Each individual Spring system is based around a microkernel known as the nucleus, which is structured to support fast cross-address-space object invocations.

This paper discusses the design rationale for the nucleus's IPC facilities and how they fit into the overall Spring programming model. We then describe how the internal structure of the nucleus is organized to support fast cross-address-space calls, including some specific details and performance information on the current implementation.

1. Introduction

Currently there is considerable interest in both industry and academia in structuring operating systems as sets of cooperating services rather than as single monolithic programs. This trend is driven by two main motivations. First, there is a belief that it will be easier (and faster) to develop and modify OS components if they are required to be cleanly isolated from one another. Second, there is a desire to blur the line between operating system components and regular applications so that third party applications can provide functionality (such as naming or paging) that was formerly thought of as part of the monolithic OS.

The Spring operating system is a distributed operating system that is focused on developing strong interfaces between OS components and on treating OS components as replaceable, substitutable parts. To aid in these goals we have chosen to represent system resources as objects and to define all our system interfaces in an object-oriented interface definition language. Additionally, the OS is structured around a microkernel, with most major system functions, such as file systems, pagers, or network software, implemented as application level services on top of this microkernel.

One of the key problems in microkernel systems is providing sufficiently fast inter-process communication (IPC), so that the OS does not suffer a major performance loss by being split into a set of different address spaces. In Spring we were interested in developing an IPC mechanism that was highly efficient and that also meshed well with our object-oriented application programming model.

This paper describes the reasoning behind our IPC model and describes how the nucleus (our microkernel) is structured so as to provide extremely efficient cross-address-space object invocation. We also provide a detailed performance analysis of our fast-path call mechanism and compare our performance with that of related work.

1.1. Short overview of Spring

Spring currently exists as a fairly complete prototype. Two components of the OS run in kernel mode. One of these is the virtual memory manager, which provides the core facilities for paged virtual memory [Khalidi & Nelson 1993A]. The other is the microkernel proper, known as the nucleus, which provides the basic primitives for domains (the Spring analogue of Unix processes) and threads.

Functionality such as file systems, naming, paging, etc. are all provided as user-mode services on top of this basic kernel. These services are provided as dynamically loadable modules and it is only at system boot time that the OS makes the final decisions about which modules are loaded onto which domains. Typically when we debug core services we will boot with a debug switch on so that all services are loaded in separate domains. In more normal use, the system start-up code will cluster sets of related services to run in the same domain.

All the inter-service interfaces are defined in our interface definition language, which supports an object-oriented type system with multiple inheritance. The system is inherently distributed and a number of caching techniques are used to boost network performance for key functions.

The system provides enough Unix emulation to support standard utilities such as make, vi, csh, the X window system, etc. [Khalidi & Nelson 1993B].

1.2. Related work

Many operating systems provide some form of message passing interprocess communication. Recent examples include sockets in Berkeley Unix [Leffler et al 1989], ports in Mach [Acetta et al 1986], and ports in Chorus [Rozier et al 1992]. It is possible to provide a procedural call and return model based on such message passing facilities but the fundamental model is of distinct threads reading messages and writing replies. In many of these system it is possible for processes to pass around access rights to communication endpoints so as to grant controlled access to given resources.

Some operating systems provide direct support for procedural IPC. Multics provided a facilities called gates that enabled secure procedural communication between different protection rings within a process [Organick 1972]. More recently, several variants of the Taos system have provided explicit cross-address-space procedure call facilities with high performance [Bershad et al 1990], [Bershad et al 1991].

The Microsoft NT system [Cutler 92] provides an interesting IPC mechanism known as event pairs that is specialized for cross-address-space calls. If a given pair of threads in different address spaces agree to use a particular event pair for communication, then one of the threads can act as a high performance server for the other thread's cross-address-space calls.

2. The Spring IPC model

2.1. Objects in Spring

Spring attempts to avoid mandating how objects should be implemented. Different applications should be able to implement objects in different ways. This includes not just the implementation of particular object methods, but also the implementation of the object machinery itself, such as object marshalling and even object invocation. For example, not all objects are required to support cross-domain calls. Objects may chose to support cross-domain calls via a variety of different mechanisms (for example using shared memory in addition to IPC). Many objects provide different mechanisms for cross-domain calls and for intra-domain calls. However, despite this ocean of flexibility, it became clear that there are certain commonly desired properties that could benefit from specific OS support.

The most important property is secure access to services. We do not want to perform a full scale authentication check whenever a client invokes a protected object. Nor do we want to restrict access to an object to a certain pre-ordained set of clients. If a client has legitimately acquired access to an object, then we would like that client to be able to pass its access on to third parties, who will then be able to operate on the object. This security requirement quickly led us to use a software capability model for providing secure access to specific objects. This approach is similar to that used in the Cambridge fileserver [Birrell & Needham 1980], Amoeba [Tanenbaum et al 1986], and Mach [Acetta et al 1986], [Sansom et al 1986].

Following from this desire for security are some secondary requirements. When a cross-address-space call occurs on a capability, neither the client nor the server should be vulnerable to the other's incompetence or malice. Thus, for example, we require that there is well defined behaviour if either the client or the server crashes. Similarly, we must be able to debug either the client or the server, without unduly disrupting the other.

Finally, there are performance considerations. Spring is a system that makes heavy use of cross-address-space calls. Thus we require a mechanism that is highly efficient, particularly for the common cases where the number of arguments and results is small.

2.2. Doors

Doors are a Spring IPC mechanism resembling in many aspects the gates mechanism of Multics [Organick 1972] or the cross-address space-call mechanisms of Taos [Bershad et al 1990], and in other aspects the communication endpoint notions of sockets in BSD Unix [Leffler et al 1989] or ports in Mach [Acetta et al 1986].

The design for doors progressed through several stages. The earliest designs were for a primitive mechanism for transferring control and data between different address spaces. A domain D1 could create a door designating a particular entry point into its address space. If this domain passed the door on to another domain D2, then a thread in D2 could jump through the door, causing it to arrive at the given entry point in D1. Doors acted as capabilities, in that a domain that obtained access to a door could pass that access on to other domains, and domains without access to a door could not fabricate access. This primitive notion of doors had an appealing simplicity. However this original design suffered two significant flaws, one major and one minor.

The minor flaw was that a door specified only a PC entry point. Since we were dealing with objects and a given server might support a wide variety of different objects, granting different access to different clients, we would normally want to be able to identify which particular object a given door granted access to. This problem could be solved by dynamically creating short sequences of code as needed to act as entry points for given objects. However, we preferred a solution that avoided the allure of fabricating code at run-time.

The major flaw was that a jump through a door implied no return path. Since we were implementing a system with an object oriented call/return model, we would normally want a server to be able to (securely) return to its callers. The obvious solution was for the caller to pass a return door along as an argument to the door call. Unfortunately, in order to sidestep some security issues caused by malicious clients potentially issuing multiple returns, we needed to be fairly careful about how we managed return doors, possibly to the point of needing to create and delete a distinct return door for every distinct call. This seemed undesirable.

Therefore, we extended the notion of doors to reflect our object oriented call and return model. A door now represents an entry point for a cross-domain call. Associated with the door are both an entry point PC and an integer datum that can be used to identify a particular object in the target domain.

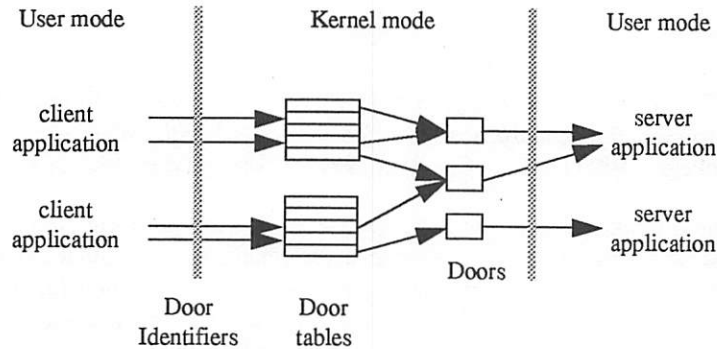
If a domain is granted access to a door, one of its threads may subsequently issue a cross-domain call through the door. This causes a transfer of control into the domain that created the door at the PC specified by the door and with the door datum in a specified register. When the nucleus executes the door call it also records some return information within the nucleus. This return information consists of the caller's domain, the PC to which to return to after the call, and the stack pointer of the caller at the time it issued the call. Then after the call has completed, the called domain will execute a door return. In executing the door return, the nucleus will use the return information that was recorded during the call and return to the callers' address space at the recorded return PC and with the recorded return stack pointer.

When a door call arrives in a server's address space, the server will typically use the datum value to locate the target object, and execute an object call. (In practice there is typically a level of indirection between the datum and actual language level objects, in order to permit dynamic interposition on objects for debugging and performance analysis.)

2.3. Door tables

Doors are pieces of protected nucleus state. Each domain has a table of the doors to which the domain has access. A domain references doors using *door identifiers*, which are mapped through the domain's door table into actual doors. A given door may be referenced by several different door identifiers in several different domains.

FIGURE 1. Doors and door tables



Possession of a valid door identifier gives the possessor the right to send an invocation request to the given door. A valid door identifier can only be obtained with the consent of the target domain or with the consent of someone who already has a door identifier for the same door. As far as the target domain is concerned, all invocations on a given door are equivalent. It is only aware that the invoker has somehow acquired an appropriate door identifier. It does not know who the invoker is or which door identifier it has used.

2.4. Reference counting

There is one last feature of doors that we should mention. Doors are typically used to provide access to application level objects such as files or I/O streams. Many servers are interested in knowing when all the customers for a given object go away, so that they can safely destroy the object and free any resources that the object is using in the server's address space. It is not sufficient for clients to issue explicit deletes on objects, as this does not cope with client crashes and it is difficult for clients to issue explicit deletes when objects are shared between domains.

Therefore we provide a simple reference counting mechanism for doors. Whenever a domain is granted access to a door, we increment the door's reference count. Whenever a domain loses access to a door (either voluntarily, or due to the domain crashing, or due to the door being revoked) we decrement the reference count on the door. When the reference count is decremented from 2 to 1, the nucleus notes that it should deliver an "unreferenced" call on the door. We use 1 rather than 0, as it is normal for an object manager to retain a reference to each door that it implements.

The nucleus keeps a queue of unreferenced doors for each domain and is prepared to use a single nucleus thread per domain to process this queue and call into the domain to deliver an unreferenced call on each of these unreferenced doors. We originally simply created threads to deliver each unreferenced call, but it is common that large numbers of doors may become unreferenced simultaneously (for example due to a client domain exiting) and having all these calls delivered in parallel caused an unnecessary resource crunch.

To avoid races between incoming calls on a door and the door becoming unreferenced, the nucleus also increments the door reference count when sending a call through a door and decrements it when the call returns. We were extremely reluctant to add this extra cost to the basic nucleus call and return code, but a scrutiny of various race conditions where an unreferenced call might arrive (causing the application to discard its object state) just ahead of a

legitimate incoming call (which tries to reference that state), convinced us that we needed to solve the problem and that we could provide a satisfactory solution more cheaply in the nucleus than in user-land.

A domain that implements a door can at any point revoke access to that door. This will invalidate all access to the door and will cause an unreferenced call to be delivered as soon as there are no longer any active calls on the door.

3. The Spring Thread Model

3.1. Threads and Shuttles

Spring is a multi-threaded system. We therefore wanted to provide application programs with ways of examining and manipulating threads. Naturally enough, threads are represented as nucleus objects, accessible via cross-domain calls into the nucleus.

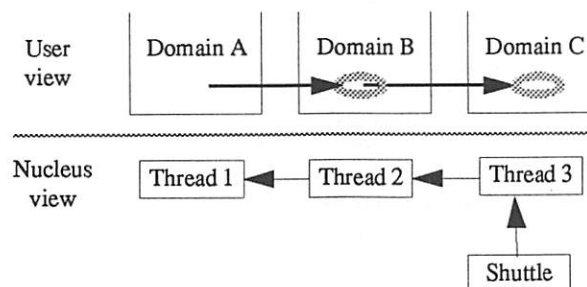
When we issue a cross-domain call from one domain into another domain we could present this to application software either as a single thread, spanning two domains, or as two threads related by a call chain. The obvious implementation is to represent a chain of cross-domain calls as a single thread. This is after all what the nucleus is really doing. However, there are two problems with this.

First, Spring is a distributed system. Network servers, known as proxies, may forward cross-domain calls to remote machines. So at the application level, several machines' nuclei may be involved in a single cross-domain call chain. Thus, if we want to represent a cross-domain call chain as a single thread, some fancy footwork would be necessary to keep the different nuclei in step. This becomes particularly interesting when one of the intermediate nuclei in a call chain crashes. (See the discussion of failure semantics in Section 3.3 below.)

Secondly, security and integrity issues mean that we want to limit the ways in which domains can manipulate one another. Say that a user domain "foo" issues a call into a file server domain. If a debugger should be attached to "foo" and attempt to stop all the threads in "foo" (a perfectly reasonable desire) then we do not want the call that is executing in the file server to be stopped — this call may be in the middle of important updates that will affect many domains in the system. Thus we wanted to limit the effects of the thread control operations to within a particular domain. If a debugger "stops" a thread in domain D1 that has called into another domain D2 then this should only affect the thread's behaviour in D1, not in D2. The call in D2 should continue executing as before; it is only when it returns to D1 that it should become stopped.

Thus, we chose to adopt a model where each application visible thread object describes a thread within a given domain. A cross-domain call chain consists of a series of these application visible threads. Access to thread objects is restricted so that an application program running on behalf of user X can normally only get access to and manipulate threads within other domains also belonging to user X.

FIGURE 2. Domains, threads and shuttles



We use the term “shuttle” to refer to the true nucleus schedulable entity that supports a chain of application visible threads. So when we execute a cross-domain call, we change the apparent application visible thread, but we retain the same nucleus shuttle, which contains the real state of the schedulable entity. (See Figure 2.)

3.2. Server threads

To service an incoming call, a domain needs to make certain resources available, most notably a stack but also such minutiae as a region for thread-local storage, a reception buffer for incoming arguments, etc. Originally we hoped to do this entirely at user-level, by having an incoming call allocate itself the necessary resources from a resource pool managed at user-level. However, this approach proved fairly cumbersome. Since several calls might arrive simultaneously, there needed to be some degree of synchronization over resource allocation. If the resource pool becomes empty, then incoming calls might either want to wait until some resources become available or try to allocate more. Doing all of this without having a stack rapidly becomes tedious.

Therefore, we decided that at the very least we wanted an incoming call to arrive in its target domain with a valid stack pointer. It also seemed desirable that all incoming calls should also be immediately associated with application visible “thread” objects so that they can be debugged and controlled.

Therefore we arranged that Spring applications can explicitly create pools of application level threads to service incoming requests. Such threads are referred to as “server threads”. When creating a server thread, the creating domain must specify a stack pointer value and a memory area to be used for incoming arguments. The nucleus will create an application visible thread object to describe this new server thread and will allocate an internal thread descriptor in the nucleus and associate it with the given domain.

So, when delivering a cross-domain call, the nucleus attempts to allocate a server thread for the target domain. If it succeeds it then delivers the data into the buffer associated with that server thread and leaps into the target domain with the previously registered stack pointer value in the stack pointer register. Unbeknown to the nucleus, the user-level thread libraries have cunningly salted away a few key values (such as a pointer to a thread private storage area) on this stack and they use these values to create a full language level environment before leaping off to a higher level object pointed to by the datum value.

Things become a little more interesting if there are no server threads available for a given process when a cross-domain call is delivered to it. We considered a couple of alternatives. We could simply return to the caller with an error code. This seemed fairly undesirable, as normally the caller will not want to have to recover from what may be a temporary overload on the server. Alternatively, we could suspend the calling thread in the nucleus until a server thread becomes available. This seems like a partial solution, and we did indeed provide this facility, but we were concerned that there might be a danger of deadlock if two domains are each waiting for a call into the other domain to complete. So we also provided a mechanism whereby a domain can find out that it has run out of server threads and, if it wishes, create more. This is done by providing an operation on a domain object where a thread can call and block until the domain runs out of server threads. In practice what the standard application libraries do is to initially avoid creating any server threads, but merely create a thread to watch for a shortfall and then create threads (up to an application specifiable limit) as needed.

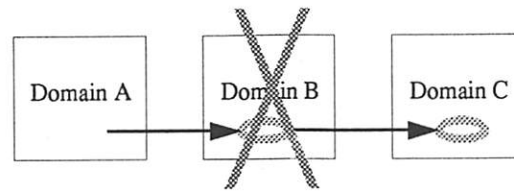
3.3. Failure Semantics

When we are executing a cross-domain call, we must be prepared to cope with either the caller or the called domain crashing. In the event of the called domain crashing the nucleus simply treats this as an involuntary return on all the active incoming calls and forces the calls to return with an error code.

Crashes by caller domains are rather more interesting. We clearly don’t want to do anything rash to affect the currently executing call, as it may be in the middle of an important activity. So we arrange that the call will return to the nucleus rather than to the true caller, so that we can remove the crashed domain from the system.

However, the called domain may be executing some lengthy activity on behalf of the caller or it may be sleeping on a resource queue (such as a tty input queue) with the intention of eventually removing some resource and returning

FIGURE 3. A crash in the middle of a call chain



it to the caller. So we would like to provide some non-intrusive way for a called domain to optionally detect that its client has gone away and thus perhaps abort whatever action it is performing.

We have chosen to use an alert mechanism similar to that provided by the Taos system [Birrell et al 1987]. Briefly, associated with each thread is a single bit specifying whether that thread is alerted. Whenever a thread sleeps it can specify that it should be awoken from that sleep with an exception if it is alerted. A thread can also poll its alerted status explicitly and set or clear it as it pleases. Or by default a thread may simply decide to ignore alerts all together.

In Spring whenever a thread becomes alerted we propagate that alert forward down its call chain to any thread it has called and to any subsequent threads. (This includes propagating the alert over the network if necessary.) Thus, when the nucleus terminates a domain, it first alerts all the threads in that domain, which will propagate an alert to any called threads. By convention, if a server thread sees an alert, it should interpret it as a mild suggestion from someone earlier in the call chain that maybe circumstances have changed and it would be wise to return home for clarification and new orders.

Crashes in the middle of call chains are particularly interesting (see Figure 3). When such a call occurs we can immediately transmit an alert to the threads downwind of the crash, but what should we do about the threads before the crash? We could wait until the call chain attempts to return into the crashed domain and then engineer an error return to the preceding domain. However this will mean that there will be an indeterminate delay until the preceding domain is notified that its call has failed. We decided to avoid this. So when there is a crash in the middle of a call chain we immediately split the call-chain into two separate chains, by creating a new shuttle to handle the call chain before the crash. This new shuttle does an immediate error return. The old shuttle, handling the call chain after the crash is redirected to return into the nucleus rather than into the crashed domain.

4. Details of door invocation

We currently support three different flavours of door invocation and door returns.

The first flavour, known as the fast-path, supports the case when all the door arguments are simple data values and total less than 16 bytes. In practice this is the dominant case for both calls and replies in Spring (see Section 5.1 below) so its performance is extremely important.

The second flavour, known as the vanilla-path, supports the case when less than 5 KBytes of data and some moderate number of doors are being passed as arguments or results. In this case the data is copied through the nucleus.

The third flavour, known as the bulk-path, supports the case when we are sending entire pages as data or results. In this case the nucleus will use VM remapping to transmit the data. The bulk-path also supports the transmission of essentially unlimited numbers of doors.

Most application level use of the door invocation mechanisms is in fact built on two higher levels of software. The first level is a mechanism known as the "transport" that provides a general purpose marshalling mechanism for storing and retrieving argument data of arbitrary size. The second level consists of C++ stubs that build on top of the

transport level. These stubs are automatically generated from our interface definition language. They provide a C++ interface to remote method operations and perform the mechanics of marshalling argument data into transports. Currently all our marshalling machinery is general purpose and is targeted to support moderate amounts of data, leading to a relatively large overhead for small calls. However since application level code only sees the interface to the C++ stubs, we plan to start generating special stubs for small calls, which will bypass the general transport machinery and leap off directly to the nucleus's fast-path mechanism.

4.1. The fast-path

The nucleus fast-path call sequence is currently roughly 60 SPARC [SPARC 1992A] instructions, and the nucleus return fast-path is roughly 40 nucleus instructions, permitting a complete cross-domain call in roughly 100 nucleus instructions. In practice there also have to be a certain number of user-level instructions at each end. On the caller, some 13 instructions are unavoidable to save and subsequently restore various pieces of user state (such as a pointer to thread-private memory, the return frame pointer, etc), since the nucleus itself saves only the absolute minimal state. Similarly, at the callee some 8 instructions are unavoidable to establish an appropriate environment for higher-level language programming and to indirect to an application-level object.

Both the fast-path call and the fast-path return only attempt to deal with the normal case where all resources are available and when neither domain has anything unusual happening to it such as being stopped for debugging or being in the process of being terminated. This is compressed into a couple of flag tests at the beginning of each fast-path call or return. If there are any problems, then the fast-path code generates a call into the vanilla-path code to handle the full work of the call or return.

4.1.1. The fast-path on SPARC V8

Clearly the details of the fast-path code will vary from machine architecture to machine architecture. On SPARC V8, one of the main considerations was managing the hardware register window set. To keep cross-domain calls cheap, we did not want to flush all the active register windows to memory when we carried out a call. On the other hand, for security and integrity reasons we wanted to avoid letting the target of a call access any register windows belonging to the client.

Fortunately the SPARC V8 architecture provides a Window Invalid Mask (WIM) that allows the kernel to prevent user-mode accessing any given register windows. So during a cross-domain call we merely mask out access to the caller's windows. As we take window overflows and write the windows out to the caller's memory, we make these windows available for the use of the called domain. A little care is required during the return sequence to ensure that the WIM is cleared correctly, and that the caller is in fact returning into the correct window.

Table 1 contains a description of the SPARC nucleus instructions required for a cross-domain call and return. The actual performance of the cross-domain path is very sensitive to cache hits and misses.

TABLE 1. Instruction breakdown of SPARC fast-path

	call	return	total
Fiddling with register windows	10	9	19
Switching thread/domain	12	11	23
Getting target info	7	0	7
Reference counting target door	3	4	7
Saving/restoring return info	6	4	10
Switching VM context	7	7	14
Miscellany	13	9	22
Total	58	44	102

To verify our understanding of this code, we obtained a full cycle by cycle trace with a hardware monitor on a SPARCstation 2. This showed no real surprises. With some minor exceptions, the CPU time is fairly evenly smeared across the different activities. On SPARCstation 2 both the trap entry and trap return sequences are fairly cheap, at about 7 cycles each, which is roughly the same as a cache miss. The actual instruction for switching the hardware VM context is slightly more expensive at about 14 cycles, but this still only constitutes a very small percentage of the overall cycle count. The main non-obvious cost is that saving the return information and writing the new thread/domain information both saturate the CPU's write buffer, causing stalls on store instructions until earlier writes are flushed.

We expect it should be possible to obtain similar instruction counts on most modern RISC CPUs. The main imponderable is the cost of switching the MMU context, which happens to be fairly low on SPARC.

4.1.2. SPARC V9.

The SPARC V9 architecture [SPARC 1992B] is a 64 bit extension of the 32 bit SPARC V8 architecture. It is fully compatible with SPARC V8 for user-mode instructions, but has a number of significant changes to the privileged mode architecture. As part of the SPARC V9 design, consideration was given to what (if anything) could be done to accelerate cross-domain calls. Fortunately, or unfortunately, the answer was comparatively little.

The cross-domain call code will benefit from several general purpose improvements to the privileged architecture that appear in V9, such as support for nested traps and the addition of an alternate set of global registers for use by trap handlers. The only changes that were specifically motivated by cross-domain calls were some changes to the register window management model to make it easier to have windows belonging to two different address spaces in the register window file simultaneously. (Basically this involved replacing a mask register with a small set of counter registers which are more convenient to manipulate, and adding additional trap vectors for different flavors of window overflows and underflows.)

Short of adding specific CISC like cross-domain call instructions (an option that was quickly rejected) we could identify no other changes that seemed likely to particularly benefit cross-domain calls. The cross domain call code is after all mostly simple, regular RISC instructions.

4.2. The vanilla-path

In the vanilla-path cases we take a full trap into the nucleus to copy the argument data into the target domain and to move any argument doors across. This code is fairly straightforward and has received no special tuning.

4.3. The bulk-path

The bulk-path is used to transfer large quantities of page-aligned data from a source domain to a destination domain. A thread in the source domain traps into the kernel using the `bulk_call` (or `bulk_return`) trap. The thread passes two arguments in registers: a pointer to a descriptors area and the number of descriptors in that area.

Each descriptor specifies an "indirect data block" that should be passed to the destination domain. Each indirect data block is a page-aligned region of the address space of the source domain. There are two ways to pass an indirect data block through the bulk path, either unmapping the pages from the address space of the source domain and mapping them into the address space of the destination domain, or by mapping the pages in both the domains using copy-on-write semantics. The nucleus uses virtual memory services [Khalidi & Nelson 1993A] to satisfy calls that go through the bulk path.

The bulk path was added recently, as an extension to the basic cross-domain call mechanism. It is currently used in the libraries to provide the illusion that a practically unlimited number of bytes and door identifiers can be transferred through the kernel.

5. Performance

5.1. Marshalled arguments size

As we mentioned earlier, our belief was that in most cases a cross-domain transfer would copy a few bytes and no door identifiers. Such transfers follow the optimized fast call and return paths. In addition we expected that in the most common cases an object invocation would follow both the fast call and return path. We based these assumptions on the structure of our interfaces and the work of other researchers [Bershad et al. 1990]. Early results show that these assumptions were correct.

In the current implementation of Spring we measured and classified the number of cross-address-space control transfers (both calls and returns) in three cases: when we boot the basic Spring services, when we "make" a medium sized C++ program and when we start an X11/OpenLook based desktop. We present the results in Tables 2 and 3.

TABLE 2. The number of calls observed in a Spring system under three different loads. (In thousands)

Cases	make ssh	start xnews	boot spring	total
fast calls	393 (84%)	91 (83%)	22 (92%)	507 (84%)
vanilla calls	75 (16%)	19 (17%)	2 (8%)	96 (16%)
total	468 (100%)	110 (100%)	24 (100%)	603 (100%)

TABLE 3. The number of returns observed in a Spring system under three different loads. (In thousands)

Cases	make ssh	start xnews	boot spring	total
fast returns	338 (72%)	77 (70%)	18 (72%)	433 (72%)
vanilla returns	130 (28%)	33 (30%)	7 (28%)	170 (28%)
total	468 (100%)	110 (100%)	25 (100%)	603 (100%)

Table 4 shows the percentage of calls and returns that passed various quantities of data. The majority of both calls and returns involved less than 16 bytes of data.

TABLE 4. Distribution of the number of bytes passed per call or return for the combined load.

number of bytes passed	percentage of calls	percentage of returns
0-16 bytes	85.0 %	81.8 %
16-100 bytes	11.5 %	13.7 %
100-1000 bytes	2.3 %	2.8 %
1000-4000 bytes	0.0 %	0.4 %
4000-5000 bytes	1.2 %	1.3 %

In addition, approximately 2.5 % of calls and 6 % of returns passed one or more kernel door identifiers.

In conclusion, our early results show that the bulk of the control transfers involve very few bytes and no doors and consequently it was worthwhile to optimize these cases in our nucleus IPC mechanism.

5.2. Performance for small calls

Table 5 shows the costs for minimal cross-address-space calls in a number of systems including Taos LRPC [Bershad et al 1990], Mach [Draves et al 1991], and NT [Cutler 1992]. All times are for bare calls on uniprocessors and exclude stubs or other higher level software costs.

TABLE 5. Minimal cross-address-space call times.

	raw time	Spec 89	scaled time
Berkeley sockets (Sparcstation 2)	850 μ s	25	21000
Taos LRPC (CVAX Firefly)	129 μ s	3.5	450
Mach 3.0/MK40 (DECstation 3100)	95 μ s	12	1100
NT event pairs (50MHz R4000)	18 μ s	70	1200
Spring cross-domain call (SPARCstation 2)	11 μ s	25	275

Since the benchmark machines vary considerably in raw CPU performance, we provide both raw times and times scaled by the CPU's Specmark89 speed. (Since we lack Specmark numbers for the exact machines used for the Taos and NT benchmarks we use Specmark numbers from product systems using the same CPU chips.) However OS performance does not necessarily scale linearly with raw CPU speed [Ousterhout 1990] [Anderson et al 1991] and the scaled numbers should be regarded as only an extremely approximate guide to relative performance.

5.3. Performance discussion

In traditional IPC systems, such as Berkeley sockets, there are three major costs:

First, there is the cost of making scheduling and dispatching decisions. Typically the thread that issued the request will go to sleep and the OS kernel will make a careful and objective decision about which thread to run next. With a little luck this will actually be the thread that will execute the request. This thread will then run and upon completion of the call it will wake up the caller thread and put itself to sleep. Once again the OS kernel will make a careful and scholarly scheduling decision and will, hopefully, run the caller thread.

Second, there is the cost of performing the context switch between the calling address space and the called address space. At its worst, this will involve a full save of the CPU registers (including floating point registers), a stack switch, and then a restoration of CPU state.

Third, there is the cost of moving the argument and result data between the caller and the callee. Traditional kernels tend to use buffer management schemes that are optimized for passing moderately large amounts of data.

Recent microkernel IPC systems have pushed back on all three of these costs. By assuming an RPC call and return model it is possible to perform a direct transfer of control between the caller and the callee threads, bypassing the scheduler. Given such a direct transfer it is also possible to avoid the full costs of a context switch and only save the minimal information that is necessary for an RPC return. By exploiting the fact that most argument sets are small (or that if they are large then they are passed through shared memory) it is possible to avoid buffer management costs.

Different systems vary in the degree to which they have succeeded in minimizing these costs. For example, the Taos LRPC system modelled a cross-domain call as a single thread of execution that crossed between address spaces, thereby avoiding and dispatching or scheduling costs. However both Mach and NT model the callee threads as autonomous threads which simply happen to be waiting for a cross-process call. This leads to a certain amount of dispatcher activity when a cross-process call or return occurs.

The Spring nucleus has attempted to minimize all three costs. The nucleus's dispatcher works in terms of shuttles, which do not change during cross-domain calls. There are therefore no scheduling or dispatching costs during a

cross-domain call. Only an absolutely minimal amount of CPU state is saved (basically a return program counter and stack pointer). We do not attempt to save the current register window set, or attempt to switch to a different kernel stack. The fast-path is optimized for the case where all the arguments are passed in registers or shared memory, so the nucleus need not concern itself with buffering arguments or results. In addition, the fast-path code is executed directly in low-level trap handlers and avoids the normal system call costs.

A more mundane factor in our IPC performance is that our nucleus data structures have been tailored to optimize their performance for cross-domain calls. For example, during a cross-domain call there is no need to check that the target door identifier is valid. Instead a simple mask and indirect load is performed. If the target door identifier was invalid we will get a pointer to a special nucleus door entry point which always returns an error code. Similarly there was an effort to concentrate the number of flags that the fast-path call and return code would need to test into a single per-thread "anomalous" flag.

If we were prepared to ignore security or debugging issues we could probably shave several more microseconds of our fast-path time. For example, we have to pay several instructions to prevent the callee thread tampering with register windows belonging to the caller thread. Similarly during both call and return we are prepared to cope with threads that have been stopped for debugging. However, for our desired semantics, we believe we are fairly close to the minimum time required for a cross-domain call.

6. Conclusions

We have provided a high performance cross-address-space communication facility, doors, which efficiently supports the object oriented communication required for the Spring operating system.

7. Acknowledgments

We would like to thank Marcel Janssens for obtaining the low level timing measurements described in Section 5.1.1, and Yousef Khalidi for implementing the Spring VM support for the bulk-path data transfers.

8. References

- [Acetta et al 1986] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young. "Mach: A New Kernel Foundation For UNIX Development." Summer USENIX Conference, Atlanta, 1986.
- [Anderson et al 1991] T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska. "The interaction of Architecture and Operating System Design." Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991.
- [Bershad et al 1990] B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems 8(1), February 1990.
- [Bershad et al 1991] B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy. "User-Level Interprocess Communication for Shared Memory Multiprocessors." ACM Transactions on Computer Systems 9(2), May 1991.
- [Birrell & Needham 1980] A. D. Birrell and R. M. Needham. "A Universal File Server." IEEE Transactions on Software Engineering, 6(5), September 1980.
- [Birrell et al 1987] A. D. Birrell, J. V. Guttag, J. J. Horning and R. Levin. "Synchronization Primitives for a Multiprocessor: A Formal Specification." Proceedings of the 11th ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987.
- [Cutler 1992] D. N. Cutler. "NT." Presentation at the USENIX Workshop on Micro-kernels and Other Kernel Architectures, Seattle, April 1992.
- [Draves et al 1991] R. P. Draves, B. N. Bershad, R. F. Rashid and R. W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems." Proceedings of the 13th ACM Symposium on Operating Systems Principles, Pacific Grove, California, October 1991.

- [Khalidi & Nelson 1993A] Y. A. Khalidi, M. N. Nelson. "The Spring Virtual Memory System." Sun Microsystems Laboratories Technical Report SMLI 93-9, March 1993.
- [Khalidi & Nelson 1993B] Y. A. Khalidi, M. N. Nelson. "An implementation of Unix on an object-oriented operating system." Proceedings of the Winter USENIX Conference, San Diego, January 1993.
- [Leffler et al 1989] S. Leffler, M. McKusick, M. Karels and J. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System." Addison-Wesley, 1989.
- [Organick 1972] E. I. Organick. "The Multics System: An Examination of Its Structure". The MIT Press, Cambridge, Massachusetts, 1972.
- [Ousterhout 1990] J. Ousterhout. "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" Proceedings of the Summer USENIX Conference, Anaheim, June 1990.
- [Rozier et al 1992] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillermon, F. Herrman, C. Kaiser, S. Langlois, P. Leonard and W. Neuhauser. "Overview of the Chorus Distributed Operating System." Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures, Seattle, April 1992.
- [Sansom et al 1986] R. D. Sansom, D. P. Julin and R. F. Rashid. "Extending a Capability Based System into a Network Environment." SIGCOMM '86 Symposium On Communications Architectures & Protocols, Stowe, Vermont, August 1986.
- [SPARC 1992A] SPARC International. "The SPARC Architecture Manual Version 8." Prentice-Hall, 1992.
- [SPARC 1992B] SPARC International. "The SPARC Architecture Version 9." 1992.
- [Tanenbaum et al 1986] A. S. Tanenbaum, S. J. Mullender and R. van Renesse. "Using sparse capabilities in a distributed operating system." Proceedings of the 6th IEEE International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986.

Author Information

Graham Hamilton is a Senior Staff Engineer at Sun Microsystems Laboratories, where he is the project lead for the Spring operating system project. His interests include distributed computing, object-oriented systems, and operating systems. He has a Ph.D. in Computer Science from the University of Cambridge.

Panos Kougiouris is currently a Member of Technical Staff at Sun Microsystems Laboratories. Before joining Sun he worked on the Choices project, a pioneer object-oriented operating system at the University of Illinois at Urbana-Champaign. His interests include distributed software, networks and architecture. He received a M.S. in Computer Science from the University of Illinois at Urbana-Champaign in 1991 and a Diploma in Computer Science and Engineering from the University of Patra, Greece in 1989. Panos can be reached at Sun Microsystems Laboratories Inc., 2550 Garcia Ave., MTV29-112, Mt. View, CA 94043, USA, or via e-mail at panos.kougiouris@sun.com.

Trademarks

Sun and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories, Inc. All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc. SPARCstation is licensed exclusively to Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

“Stacking” Vnodes: A Progress Report

Glenn C. Skinner and Thomas K. Wong

SunSoft Inc.
2550 Garcia Avenue
Mountain View, Ca 94043

ABSTRACT

People are dissatisfied with the file system services that come with their UNIX systems. They want to add new and better features. At present they have two choices: express their service as a user-level NFS server, or use the vnode/VFS interface to build at least part of it into the kernel. Although the vnode/VFS interface has been remarkably successful as a kernel structuring concept, it has failed to provide source portability between UNIX versions or even binary compatibility between releases of the same UNIX version. It has been obvious for some time that a redesign of the vnode/VFS interface that allowed file systems to be shipped as binary kernel modules that survive from release to release is needed. We describe a prototype kernel with a vnode/VFS interface that would allow this. It is based on earlier work on “stacking” vnodes at Sun and at UCLA, but it replaces the stacking concept by a more strictly object-oriented concept of interposition.

1. Introduction

People are dissatisfied with the file system services that come with their UNIX systems. Existing file systems are looking increasingly old and shopworn, and developers want to add new and better features. At present they have two choices: express their service as a user-level NFS server, or use the vnode and Virtual File System (VFS) interfaces [Kleiman 86] to build at least part of it into the kernel. Neither of these approaches is satisfactory.

User-level NFS servers are a viable approach for some purposes, and people have used them to implement services such as automounters [Callaghan & Lyon 89, Pendry 89], autocachers [Minnich 93], and the Restore-o-Mounter [Moran & Lyon 93]. These services all primarily concern themselves with name space manipulation and stay out of the way of data movement. Indeed, the big drawback of user-level NFS servers is that it is nearly impossible to provide adequate performance and semantics for data movement operations. This limitation severely restricts the approach’s applicability.

Kernel-resident file system modules avoid the limitations of user-level NFS servers, but suffer from problems of their own. They rely critically on the vnode/VFS interface as their means of connection into the kernel environment. But the interface suffers from problems of compatibility and stability. [Webber 93] states:

UNIX kernels with a VFS architecture have been commercially available for many years. Sun Microsystems, for example, described their VFS architecture in the 1986 Summer USENIX proceedings. By many measures the VFS concept has been quite successful, but from a third party point of view there are two major problems: few vendors have the same VFS interface [and] few vendors provide release-to-release source or binary compatibility for VFS modules.

We call these two problems the *VFS portability* problem and the *lock-step release* problem, respectively. Together, they make VFS modules expensive to produce, expensive to port, and expensive to maintain.

Compounding these problems is the interface's lack of modularity: complete file system implementations are large and complex, and the vnode/VFS interface provides no support for decomposing them into smaller components.

It has been obvious for some time that a redesign of the vnode/VFS interface that addressed these problems is needed. We describe a prototype kernel with a vnode/VFS interface that allows for binary compatible, modular file system implementations. It is based on earlier work on "stacking" vnodes at Sun and at UCLA, but it replaces the stacking concept by a more strictly object-oriented concept of interposition.

In the rest of this paper, we start by analyzing the deficiencies of the existing interface. We then introduce the interposition model and explain how it addresses these deficiencies. In subsequent sections we describe the design of a prototype system that realizes this model and our experiences implementing the prototype. Next we go over a laundry list of problem areas we have not yet addressed. Finally, we examine related work and present concluding remarks.

2. Problems with the Vnode Interface

The Virtual File System (VFS) and vnode interfaces form the boundary between file system implementations and the parts of the kernel that access file system services. The VFS interface is concerned with operations pertaining to file systems as a whole, such as mount and unmount, whereas the vnode interface governs access to individual files. The two interfaces have a similar structure, so the discussion below concentrates on the vnode interface as illustrative of both. Figure 1 shows an idealized sketch of the vnode interface.

```
struct vnode {
    struct vfs      *v_vfsp;      /* owning VFS */
    enum vtype      v_type;      /* file, dir, etc. */
    struct vnodeops *v_op;        /* the ops vector */
    void            *v_private;   /* internal state */
    void            *v_public;    /* visible state */
};

struct vnodeops {
    int  (*vop_open)();
    int  (*vop_read)();
    int  (*vop_write)();
    /* and so on */
};
```

Figure 1: Standard Vnode Interface

The interface is object-oriented in the sense that it represents each file as a data structure called a *vnode* along with a set of operations (a so-called *ops vector*) for manipulating it. Every vnode belongs to some VFS, which is responsible for defining its operations and maintaining its internal state. Clients of the interface invoke vnode operations through macros like:

```
#define VOP_X(vp, args)  (*(vp)->v_op->vop_x)(vp, args)
```

In object-oriented terminology, calling this macro requests the vnode object **vp* to invoke its *vop_x* method.

In the introduction, we indicted the vnode interface on two basic counts: that it lacks stability and that it lacks modularity. Both of these failings are aggravated by non-object-oriented aspects of the interface.

- Vnodes are not opaque.

Some of the vnode's internal structure is visible and available for direct manipulation by code outside of its owning VFS. This exposure makes it very difficult to change the vnode structure without breaking binary compatibility and is a major contributor to the interface's instability from release to release. It also complicates resolving the next weakness:

- The vnode interface makes no provision for inheritance.

From the interface inheritance viewpoint, the interface is not extensible, making it difficult to accommodate file systems with novel semantics. From the implementation inheritance viewpoint, the interface is

not modular, making it impossible to develop file systems as incremental units of behavior added to previously available components.

3. Interposition and Composition

The deficiencies listed in the last section lead directly to a set of requirements for a revised vnode interface. The interface should be object-oriented. Specifically, it must:

- accommodate multiple file systems implementing the interface,
- be opaque, and
- support some form of implementation inheritance.

Another desirable requirement, that our work does not address, is that the interface should:

- support some form of interface inheritance.

The existing vnode interface already satisfies the first requirement. The second is straightforward: in many cases existing references to exposed state turn out to be unnecessary and can be eliminated. We handled the remaining references by converting them to calls to access functions, which we added to the set of vnode operations.

The implementation inheritance requirement is trickier. We require that the interface allow for binary compatibility of independently developed modules, which rules out language-based subclassing schemes. One way to satisfy this requirement, and the way we chose, is to use delegation rather than strict inheritance. This technique allows one file system module to forward an operation request (possibly modified) to another module as part of that operation's implementation.

Experience with a previous attempt to add delegation to the vnode interface [Rosenthal 90] and contemplation of its aftermath [Rosenthal 92] had alerted us to a pitfall in adding delegation to the interface. In the existing vnode interface, a given vnode serves two purposes: it is the in-kernel representative of some file system object, and it records the state and operations that the file system implementation applies to that object. That is, vnodes capture both identity and behavior. We wanted to use delegation to add behavior, but not at the price of adding unwanted identities. This observation led us to use a specialized form of delegation that we call *interposition*, where a delegating (or *interposing*) file system module in effect takes control of the target of the delegation, by capturing all accesses to it.

Putting these requirements together led us to the revised vnode interface outlined in Figure 2.

```
struct vnode {
    struct pvnode *v_chain;      /* head of interposition chain */
};
struct pvnode {
    struct vfs *p_vfsp;         /* owning VFS */
    struct vnodeops *p_op;      /* the ops vector */
    struct pvnode *p_link;     /* next older interposer */
    void *p_private;           /* this interposer's state */
}
struct vnodeops {
    /* as before */
};
```

Figure 2: Interposition Vnode Interface

We retain the term *vnode* for the part that captures the identity of the corresponding file system object. A *pvnode* contains the information that describes the behavior that a given file system imposes on an object; each *pvnode* belongs to a particular file system implementation as recorded in the *p_vfsp* field and is attached to a single *vnode*. *Pvnodes* are organized into *interposition chains*; the *vnode*'s *v_chain* field names the first, most recently interposed, *pvnode* in its chain and the *p_link* *pvnode* field links to the next most recent interposing *pvnode*.

We illustrate how interposition works through an example.

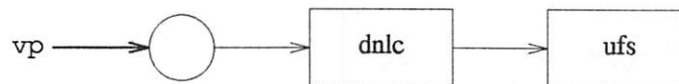


Figure 3: An Interposition Chain

Figure 3 depicts a vnode named by `vp` and its associated interposition chain; the circle designates the vnode and the boxes its interposing pvnodes. Suppose the file associated with `vp` is a directory and some process tries to look up a file in it. The kernel's vnode layer will issue the call `VOP_LOOKUP(vp, NULL, name, &resultvp)` to initiate the lookup. `VOP_LOOKUP` is defined as:¹

```

#define VOP_LOOKUP(vp, pvp, name, result) vn_lookup(vp, pvp, name, result)
int vn_lookup(struct vnode *vp, struct pvnode *pvp, char *name, struct vnode **result)
{
    struct pvnode *npvp = pvp ? pvp->p_link : vp->v_chain;
    return (*(npvp->p_op->vop_lookup)(vp, npvp, name, result));
}
  
```

Since the initial call has a `NULL` pvnode argument, control passes to the lookup method defined as part of `dnlc`'s ops vector. `Dnlc` is an interposer that the directory name lookup cache VFS module defines. Its lookup method consults the module's cache to see whether the target file already has a vnode. If so, the method sets the `result` argument to that vnode and returns. Otherwise, it forwards the operation to the base `ufs` file system module with `VOP_LOOKUP(vp, pvp, name, result)`, where `pvp` is the pvnode argument it was called with. When the forwarded call returns, `dnlc`'s lookup method enters the resulting vnode into the cache and returns.

The preceding example showed how one can build up behavior applicable to a single object. However, there is often need to combine multiple objects together into a composite whole. This process is called *composition*; the interposition framework accommodates it by allowing pvnodes to maintain references to other vnodes as part of their private state.

A reference to the composing vnode acts as an entry point into the overall composite object. The components of the composition may be named by referring to their vnodes; it is up to the composer to ensure that such accesses are semantically sensible, perhaps by interposing on each of the composees to exert control over their behavior.

Here is another example, showing how to combine composition with interposition to handle file system mounts. We use the technique illustrated in this example to factor knowledge of mount point semantics out of all our file system implementations.

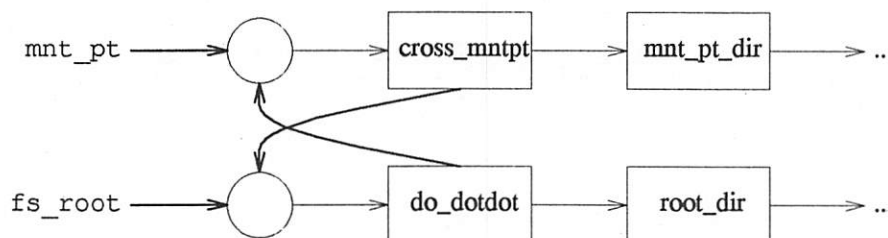


Figure 4: Mounted File Systems

In Figure 4, `mnt_pt` is a vnode naming a directory acting as a mount point and `fs_root` names the root of the file system mounted there. `Cross_mntpt` and `do_dotdot` are interposers that supply modified versions of

¹ This definition is simplified for the sake of illustration; locking considerations and the need to determine whether a new vnode has been created make the actual definition more complicated.

VOP_LOOKUP and pass all other operations through respectively to *mnt_pt_dir* and *root_dir*; they in turn head the remainder of *mnt_pt*'s and *fs_root*'s interposition chains. *Cross_mntpt* and *do_dotdot* each compose on the other vnode participating in the mount.

When a lookup request reaches *mnt_pt*, control first reaches *cross_mntpt*. Its lookup method compares the name argument with *".."*. If they are equal, this component of the overall path name being looked up passes out from underneath the part of the file system name space the mount controls; otherwise, the component lands inside the mount. *Cross_mntpt*'s lookup method proceeds accordingly, by forwarding the request to the next interposer (*mnt_pt_dir*) if the name is *".."*, and by passing the request on to its composee (*fs_root*) otherwise. *Do_dotdot*'s lookup method is the dual of *cross_mnt_pt*'s; if the name is not *".."*, it forwards the request to the next interposer and otherwise passes the request to its composee.

4. Design

We wanted to limit the scope of our prototyping effort, so that the project remained tractable. Thus, we decided not to tackle adding support for transactions or page cache consistency to the vnode interface, since we knew from previous work that they pose tough problems. Instead, we decided to see how much mileage we could get from developing our ideas about interposition. We knew at the outset that we would have to solve problems relating to vnode configuration and locking. Not surprisingly, we also encountered problems we had not anticipated. We discuss some of them below.

4.1. Vnode Configuration

When the system creates a new vnode it must be configured with interposers that properly capture the semantics desired of the corresponding file system object. Thus, operations such as VOP_LOOKUP that potentially create new vnodes must have hooks for distinguishing newly created vnodes from previously existent ones and for adding behavior (new interposers) to new vnodes.

We satisfy this requirement by having the vnode layer supply a virgin vnode as a tentative candidate to be returned as the result of the operation. If an interposer decides that an existent vnode is appropriate as the return value, it is free to substitute that vnode for the tentatively supplied one. If an interposer needs to tell whether a forwarded operation has returned a new vnode, it can check to see whether the forwarder returned the same vnode it was given.

Now we have the tools we need. To ensure that the vnode resulting from a lookup call behaves properly, we arrange for the directory vnode handling the lookup call to have an interposer that examines the result vnode. If the vnode is new, the interposer pushes whatever VFS modules are needed to supply the requisite behavior. If not, the interposer leaves the result vnode alone, since it presumably had instances of the proper modules pushed when it was originally created.

4.2. Concurrency and Locking

We wanted our implementation to allow maximal concurrency for operations on a given vnode and, in particular, to allow pops (removing the most recent interposer) to proceed in parallel with other operations. We also needed a way to ensure atomicity of creates and destroys, so that intermediate states occurring as behavior is added or removed are invisible.

Vnode destruction posed particular problems. The implementation destroys a given vnode by popping off interposers until none are left and then deallocating the vnode itself. During this time a concurrent VOP_LOOKUP could retrieve an as yet unpopped pvnode from the defunct vnode and attempt to return that vnode. We had to design a locking protocol between the interposition framework and file system implementations that prevented this race condition and was robust in the face of deallocating the vnode while the file system was attempting to synchronize with it.

Our design uses a single lock in each vnode to guard that vnode's interposition chain. Multiple threads can acquire the lock for shared access, or a single thread can acquire it for exclusive access. A given thread can acquire the lock recursively in either access mode, although an attempt to upgrade from shared to exclusive access will block until no other threads hold the lock. Whenever a thread executes a vnode operation, the interposition framework acquires the vnode's lock on its behalf. While holding the lock, the framework obtains a

reference to the pvnode supplying the method to be executed and updates a count of threads executing in that pvnode.

When initiating operations that can cause plumbing changes, such as `VOP_POP` or `VOP_LOOKUP`, the framework acquires the exclusive version of the lock, to prevent other threads from accessing the vnode while it is in an inconsistent state. For destruction operations, the framework locks the vnode being dismantled; for creation operations, it locks the newly created vnode. In either case, the thread executing the operation can go on to initiate other operations on the target vnode, since it holds the exclusive version of the lock and it is legitimate to acquire that lock recursively.

We handle destroy/lookup races between the framework and VFS modules by breaking the operations up into phases. When popping a pvnode, the framework notifies the pvnode's VFS by calling `VOP_POP`. This call informs the VFS that the pvnode is no longer attached to its vnode, although there may still be threads executing within it or in other pvnodes further down the interposition chain. At this point, the VFS must update the pvnode's internal state to mark it as having been popped and henceforth refrain from trying to discover what vnode it is attached to. When no more threads are executing in the pvnode (the reason for maintaining the thread execution count), the framework calls `VOP_INACTIVE` to tell the owning VFS that it can destroy the pvnode.

On the lookup side, a VFS that finds a pvnode that would satisfy the lookup request must first mark it as a candidate for use, to prevent the inactive routine from destroying it. The VFS must then check to see whether the pvnode has been popped but still has threads executing against its old incarnation. If so, it must wait for the threads to finish and then recheck its lookup criteria. Otherwise, it asks the interposition framework to retrieve the vnode that the pvnode is attached to. This request may fail because of a concurrent `VOP_POP` or `vn_inactive` call; if so, the VFS must try again after handshaking with the other thread. Finally, after successfully having run this gauntlet, the VFS's lookup method can return the vnode.

4.3. Ill-Defined Vnode Operations

Before starting to work on the design, we had realized that the interposition model's strong object-orientation implied that it must be possible to designate a single vnode as the recipient of each operation, but we failed at first to appreciate all of the ramifications. We soon ran into trouble with operations, such as `VOP_CMP` and the directory operations (`VOP_LINK`, `VOP_RENAME`, et al), that act on multiple vnodes.

In the old system, two vnodes from the same VFS exhibit the same behavior because they inherit the same vnode ops vector. However, this may not be true under the interposition model, since different behaviors may have been pushed onto either one or both of the vnodes. An immediate consequence is that we don't know what to do with these vnode operations under the interposition model, since the choice of vnode to receive an operation may dictate its results. (E.g., the result of `VOP_CMP(v1, v2)` may not be the same as `VOP_CMP(v2, v1)`.) More fundamentally, it is only possible to interpose on one of the vnodes involved in the operation (the one to which the operation is directed), but a basic tenet of the model is that interposers have an opportunity to influence all aspects of the behavior that the corresponding vnode exhibits. Hence, operations that use secondary vnode arguments for anything other than their identity are incompatible with the interposition model.

We addressed this problem by decomposing each vnode operation that has multiple vnodes as input arguments into primitive vnode operations that each only require one vnode as an input argument. For example, `VOP_LINK` turned into calls to the file vnode to fetch its file identifier (fid) and to increment its link count, and a call to the directory vnode to add an entry associating the fid and leaf name.

These decomposed operation sequences have to be atomic; it would be incorrect to allow other threads to observe intermediate states occurring part way through a sequence. One obvious response to this requirement would be to introduce transactions. We were reluctant to take that step and were able to avoid it by exploiting properties of the vnode lock we had already introduced to maintain interposition chain integrity.

When the vnode layer gets a multiple-vnode operation it starts by determining which vnodes will participate in the operation. It sorts them into a canonical order (by address) to avoid deadlock, and acquires each vnode's exclusive lock. At this point, the thread handling the operation can do whatever is required to the affected vnodes without concern that intermediate states will be visible to other threads. It proceeds by

executing the same algorithm that the underlying file system implementation would execute in the old system, calling the decomposed single-vnode methods to execute each step. After completion, the vnode layer unlocks the vnodes, and the modified state resulting from the overall operation becomes visible.

The vnode layer has to take care to handle errors from intermediate operations by issuing operations to back out of the changes made so far; it maintains enough state to allow it to do so. There is no guarantee that these back-out operations will succeed; this situation is one where our pseudo-transaction technique has failure modes that the old system does not exhibit.

The current set of decomposed vnode operations is listed below in Table 1. We do not regard it as final and may change it as our design evolves.

Table 1: Decomposed Vnode Operations

Name	Purpose
VOP_FID	get a file's fid (file identifier)
VOP_MKOBJ	create an object (e.g., regular file, directory, special file, symbolic link) and return its vnode and fid
VOP_DIRADDENTRY	add a directory entry for the file identified by the fid
VOP_DIRRMENTRY	remove a directory entry
VOP_INCLINK	increment a file's link count
VOP_DECLINK	decrement a file's link count

4.4. Transforming VFS Operations into Vnode Operations

To operate correctly under the interposition model, many VFS operations also need the ability to interpose on other VFSs. For example, the (old) `VFS_SYNC` call must be forwarded to every VFS that is part of a given mounted file system to insure proper ordering when flushing data to the file system media. To reduce complexity, we would like to avoid interposing on VFS operations. To achieve this end, we have converted many VFS operations into vnode operations. The underlying idea is to treat a VFS operation as a vnode operation for which the file system(s) interpreting the operation ignore the identity of the vnode to which the operation was issued. In other words, a VFS operation can go to any vnode associated with the target VFS.

Table 2 lists the set of VFS operations we have transformed into vnode operations.

Table 2: Vnode Operations Transformed from VFS Operations

Name	Purpose
VOP_PREPARE_UMOUNT	check if the VFS identified by the vnode is ready to unmount (e.g., not busy)
VOP_UMOUNT	unmount a VFS identified by the vnode
VOP_VGET	return the vnode identified by the <i>fid</i>
VOP_STATVFS	return <i>statvfs</i> information for the VFS identified by the vnode
VOP_SYNC	flush data belonging to the VFS identified by the vnode to the underlying media

5. Implementation

Having resolved the design issues described above, it was time to start implementing our prototype. We soon had a key insight: that it was possible to have the interposition framework coexist with the old vnode layer, with all the vnodes associated with a given mount point being uniformly either old style or new. This idea allowed us to develop our prototype incrementally, first converting to the interposition framework simple file systems such as *fdfs* and *tmpfs*, and later more complex file systems such as *ufs* and *specfs*.

For our prototype to be a successful proof of concept, we thought it important for it to boot using new style mounts as the default. We have converted it to do so, although some file systems such as *nfs*, *fifofs*, and *procfs* remain available only on their old form.

Our strategy of allowing old and new style file systems to coexist worked according to plan. This technique has allowed us to obtain valuable performance information without having to convert all file systems to be interposition-aware. We also view it as the basis of a migration strategy for converting to the interposition framework; file system developers can choose to take the plunge at their own pace without facing a system-wide flag day.

5.1. VFS Conversion Experience

The first step of our implementation was to change the kernel to allow old and new style file systems to coexist. We did this by redefining the vnode structure to be the union of the old version and the new. We added a new `VOP_GETVNVER` operation to return the vnode structure's effective version number. As mentioned above, we also added a set of new vnode operations (e.g., `VOP_GETVRDEV`) to act as access functions for previously accessible fields in the old vnode structure that are no longer visible (e.g., `v_rdev`). Since the ops vector field is unused for new style vnodes, we assigned a set of "just say no" operations to that field in all such vnodes, to return `ENOSYS` if invoked inadvertently. We implemented parallel sets of routines in the vnode layer to handle old and new style vnodes. The implementation switches to the appropriate routine based on the target vnode's `VOP_GETVNVER` value. Some routines, such as `lookuppn`, required more work, as they have to cope with both old and new vnodes, which can occur when crossing mount points.

The first file system we converted was *tmpfs*. The main objective was to test file object creation and directory services. We chose it first because it has no persistent storage. Therefore, it is simpler to debug, and has a smaller probability of triggering race conditions that would expose bugs in the prototype. The conversion was smooth, and produced no surprises.

Our next file system was *ufs*. Here, we looked for components that could be reimplemented as interposers and could therefore be reused by other file system implementations. This was the most difficult file system to convert, because it is more likely to encounter race conditions while waiting for disk i/o completions. In addition, the converted version had to be efficient, or else the performance goals of the prototype would not be met.

Specfs presented a new challenge because it uses composition to support devices that have multiple device names with the same major and minor device number. Since we retained the same basic design, we thought the conversion would be straightforward. This was true for block devices, but we encountered difficulties when dealing with character devices, especially with clone devices and the console device. The main problem was that the kernel directly accessed the private data of *specfs* vnodes, violating vnode opaqueness. There are two reasons why this problem occurs:

- The `VOP_GETATTR` operation is not extensible and cannot return attributes (e.g., `dev_info` and `common_specvp`) that are unique to *specfs*, but that the kernel needs. This is a deficiency of the old vnode interface.
- It is more efficient to access these attributes directly.

The ideal solution to this problem would be to provide an extensible `VOP_GETATTR` operation to return both the file system type independent and file system type specific attributes, thus allowing the kernel to access *specfs*-specific data while still respecting vnode opaqueness.

The short term solution we devised was to add a new `VOP_GETKEYEDDATA` operation, which allows interposers that recognize the input "key" argument to return interposer-specific data.

Overall, our conversion effort was not too difficult and, in fact, it became simpler as we identified more interposers and made them available. With the exception of debugging *specfs*, the toughest part turned out to be breaking down existing directory-related vnode operations into more primitive ones.

5.2. The Interposer Toolkit

One of our goals in this project is to make file system implementation easier. We plan to meet this goal by providing a toolkit of interposers that file system implementations can depend on to meet standard requirements (e.g. POSIX), to satisfy the UNIX file system semantics, and to acquire new and enhanced file system features.

As we convert existing file systems to operate under the interposition model, we try to identify file system components that can be removed and reimplemented as interposers. The set of interposers we have identified so far is listed below. We expect to extend the toolkit with new interposers as we continue our effort to factor existing file systems into smaller components.

Table 3: The Interposer Toolkit

Name	Purpose
config	push the proper behavior onto a new vnode
dnlc	add/remove a name to/from the directory name lookup cache
mfrl	handle mandatory file and record locking
mount	splice a VFS into the file system name space
specref	provide access to special files
ro	allow only read-only vnode operations to proceed

We were pleasantly surprised by how easy the interposer toolkit is to use. If certain standard file system features (e.g., *dnlc*) are desired, all it takes is to include them as part of the file system configuration at mount time. There is no need to do any programming in the file system implementation. This is a big win over the existing approach for providing common file system services through library services because the file system implementors must write code in advance in order to take advantage of these services.

5.3. Performance

We used two sets of benchmarks to assess our prototype.

The first benchmark consisted of a *perl* script used to time creation and deletion of 1000 files of different sizes (e.g., 0k, 1K, 4K, 10K) on a new style *ufs* file system that used all of the toolkit interposers listed above except *ro*. We used this test mainly during the early stage of the prototype development. The data obtained from this benchmark showed no significant performance degradation. We found these results highly encouraging, since our prototype properly handled multi-threaded locking and used interposition in a nontrivial way.

After we had converted *specfs* and switched the prototype to use new style vnodes as the default, we ran the script, and again found no significant performance degradation. We then ran one of our standard benchmarks to measure the prototype's throughput under a workload simulating a C program development environment. The system throughput from the benchmark is reported as the number of work scripts processed per hour, for various values of a workload parameter representing the number of simulated users. We expected this benchmark to reveal some amount of performance degradation. Because of the factoring that we have done, both of vnode operations and file system implementations, optimizations possible under the old monolithic implementations are no longer available.

To run this throughput benchmark successfully, a system must be very robust and our prototype has only recently reached this level of maturity.

We ran the benchmark on a Sun Sparcstation 1 with 16M memory installed with SunOS 5.1. Table 4 compares results from runs with our prototype kernel and with the standard 5.1 kernel.

Table 4: Throughput Results

Workload	SunOS 5.1	Prototype	degradation (%)
	Scripts/Hour	Scripts/Hour	
1	13.83	13.83	0
16	185.03	181.82	-1.7
24	198.58	191.28	-3.6
28	193.70	184.85	-4.5
32	175.42	179.47	+2.3
36	159.41	143.35	-9.9

The system throughput benchmark result indicates that the prototype suffers less than 5% degradation in peak throughput, but degrades more rapidly after the peak workload. We have not yet analyzed the cause of the

degradation, but suspect that it is a symptom of additional CPU overhead. Note that we have not yet tuned the prototype, and have not implemented performance saving features that are now possible under the interposition model. We therefore believe that we should be able to further improve the prototype.

6. The To-Do List

In this section we discuss areas that our prototype does not yet address. We start with things we think we understand and proceed to issues of increasing difficulty.

6.1. Configurator Interposers

If we succeed in converting existing file systems into a rich interposer toolkit of file system components, we will have created another problem: How can people specify combinations of components that do what they want? We expect that most users would prefer to remain ignorant of the issue altogether and that most others would prefer to use pre-existent file system configurations that are known to behave properly. Thus, the problem reduces to devising a way for file system vendors to specify "canned configurations".

In the existing system, each mount can differ by file system type from other mounts and the mount system call takes an argument specifying that type. We plan to exploit this interface to use the type argument to name a *configurator* interposer that is responsible for ensuring that all files underneath the new mount point behave appropriately. The mount system call will push this interposer on the root vnode of the newly mounted file system. It is subsequently responsible for pushing a copy of itself on directories, as lookup calls create vnodes for them, and for pushing the sequence of interposers that makes up the canned configuration on all files, as lookups create new vnodes for them. By induction, the overall effect is to ensure the proper layering configuration for all vnodes associated with the mount.

6.2. A Directory Interposer

We want to split *ufs* into a directory piece and an inode-level file access piece, and make the former available as an interposer that other base file systems can use. With the old set of vnode operations, this separation is infeasible; operations such as `VOP_CREATE` mix together actions from both pieces (creating a file and entering it in a directory), preventing their separation. Fortunately for this purpose, we have already had to decompose operations that mix directory access with file access, as explained above.

When pushed on top of a regular file, this interposer will make it behave as a directory. For example, it will handle `VOP_LOOKUP` calls by reading the underlying file, looking for a matching entry. If it finds one, it will extract the fid from the entry and use it as the argument to a `VOP_VGET` call on the file. The VFS defining the file's behavior will then look the fid up and return its vnode, which the directory interposer then returns in turn. Other operations proceed similarly.

Note that the directory interposer expects the VFS supplying behavior for the file it is pushed on to supply a primitive directory service of its own, in the form of the `VOP_VGET` operation. The interposer converts that VFS's flat, name-by-fid name space into the standard hierarchical name space.

6.3. Impedance Mismatches

Decomposing multi-vnode operations into more primitive single-vnode operations has consequences for existing file systems. Consider, for example, the NFS client side implementation. It has a particular fixed vocabulary, in the form of the NFS protocol, that it must use to communicate with the server. This vocabulary is well-matched to the old set of vnode operations, but is at a considerably higher level than the new set of operations. Thus, the client implementation now faces an impedance matching problem where it must convert an incoming stream of vnode operations into a corresponding stream of NFS protocol RPCs. Because of the different levels of the two streams, the client implementation must accumulate incoming vnode operations until they add up to a protocol operation.

We can handle this new implementation requirement by adding to the implementation a state machine that tracks progress in accumulating operations that add up to a protocol request. However, the mechanics of the addition are likely to be ugly. One complication is that operations can occur concurrently on multiple vnodes, and the implementation must keep these logically separate input streams distinct.

We plan to address this problem by taking a small step toward adding transactions to the vnode interface. Our motivation is the observation that the vnode layer now handles many incoming requests by decomposing them into more primitive operations. Thus, it has knowledge of which operations are associated with which higher level requests. Stated differently, the vnode layer is effectively creating transactions, so it might as well pass some information about them to the participants. Thus, we intend to add an additional argument to each vnode operation identifying the transaction to which the operation belongs.

6.4. Transactions

Having tiptoed this far toward full transaction support, why not go the whole way? We have a design sketch [Rosenthal 92], but are reluctant to use it because of concerns about its ramifications. Adding transaction support to the kernel's vnode layer and to file system implementations is as invasive a step as converting them to the interposition model. We judged that making two changes of this magnitude would add unacceptable risk to our project.

However, the project is now at a point where it is time to reexamine this decision. If we decide to proceed, our transaction facility would be confined to the kernel's vnode layer and to file system implementations; it would not be available at user level for general use. We would use it to replace our current ad hoc lock-based implementations in those parts of the prototype.

Here is a sketch of the transaction design and the effect of using the facility on some sample code. Each vnode operation would take an additional transaction context argument (replacing the transaction identifier mentioned in the previous section). If the context is `NULL`, the operation is not part of a transaction and proceeds as before. Otherwise, the operation must join the transaction the context argument identifies, by adding a record of its own to the context. If the operation cannot join the transaction, it must fail, causing the transaction to abort. Each record in the transaction context contains pointers to callback routines to be executed when the transaction aborts, prepares to commit, or commits. The transaction framework provides routines to invoke the appropriate callback from each record, for use when the transaction reaches the abort, prepare, or commit stage.

After restructuring to use transactions, the code in Figure 5 becomes that of Figure 6.

```

if (VOP_FOO(vp) != 0) {
    /* About a page of code cleaning up the mess */
    goto bad1;
}
/* The FOO operation succeeded - do something more */
if (VOP_BAR(vp) != 0) {
    /* About a page of code cleaning up the mess */
    goto bad2;
}
/* The BAR operation succeeded - do something more */
return (0);
bad2:
bad1:
return (ENOTTY);

```

Figure 5: Code Before Transactions

6.5. Page Cache Consistency

Up to now, we have avoided grappling with the interaction of the VM system with the interposition framework.

A major issue is how to structure the page cache. In the old system, each vnode has an associated list of pages whose contents cache parts of the corresponding file. The pages are each marked with the address of the vnode and offset into the file. As a matter of philosophy, we believe that each VFS module should be able to interact with the page cache as it sees fit to store data consistent with the behavior it imposes on a given file. This consideration suggests that the page cache should identify pages by pvnnode and offset instead of by vnode and offset, as in the old system.


```

vtc = kmem_alloc(sizeof struct vtc);
if (VOP_FOO(vp, vtc) == 0) {
    if (VOP_BAR(vp, vtc) == 0) {
        if (vtc_try_commit(vtc) == 0)
            return (0);
    }
}
return (vtc_abort(vtc));

```

Figure 6: Code After Transactions

The other primary issue is how interposers and composers that cooperatively interact to manage a composite file can maintain coherence among the pages that they have entered into the cache while still providing for efficient i/o involving those pages.

We have preliminary ideas about how to resolve these issues, but they are not yet well enough developed to report. However, it is already apparent that we will have to add a richer set of methods for file system modules and the VM system to use to communicate with each other; anything that the VM system does that might affect the status of a cached page must be visible to the relevant file system modules and vice versa.

7. Related Work

People have taken two basic approaches to overcoming deficiencies in the system's file system infrastructure: finding better ways to support user level file system modules and attacking the limitations of the vnode interface.

Watchdogs [Bershad & Pinkerton 88] and the File Monitor Interface [Webber 93] are examples of the former approach. Both are founded on the premise that the kernel is a sufficiently inhospitable environment that it is desirable to provide facilities for user-level file system modules. In contrast to the user-level NFS server approach, they rely on kernel modifications and a specialized user-kernel communication channel to provide finer control over how the user level file system process can handle operations, both in the choice of which operations require intervention and in how to dispose of them. In contrast with in-kernel techniques such as the interposition model, approaches that attempt to move most of the work to user level offer a congenial and portable file system development environment, at the price of context switching overhead and the possibility of deadlock and additional data copying overhead. They are also potentially compatible with the interposition approach; the kernel hooks for communicating with a user-level process could be packaged as an interposer that is pushed onto vnodes of interest.

The Ficus project at UCLA [Guy et al 90, Heidemann 92] and the stacking vnode work of [Rosenthal 90] pioneered the other major approach to improving the file system infrastructure. Both extend the vnode interface by introducing facilities for vnode "stacking".

Heidemann's Ficus work added interface inheritance to the vnode interface through operation extensibility: file system modules are free to define new vnode operations. At system boot time, the layering framework determines the complete set of vnode operations and constructs ops vectors for each file system module. His framework also supports static implementation inheritance; file systems are stackable in the sense that their vnodes can compose on those of other file systems. The layering configurations for vnodes of each mountable file system type are determined at system boot time and remain fixed for each vnode's lifetime. This work has been successful enough to merit incorporation into the Berkeley 4.4bsd release.

Our interposition work is a direct descendant of Rosenthal's stacking vnode prototype [Rosenthal 90]. His system supports implementation inheritance in the form of dynamic composition, by forming vnode stacks. The configuration of a stack may change dynamically over its lifetime. His layering framework provides explicit support for stacking in that, unless a file system module explicitly chooses otherwise, operations directed at any of the vnodes in a stack are redirected to the vnode at the top of the stack. This mechanism guarantees that stack configurations are opaque to their clients, but also prevents a vnode from being part of more than one stack, precluding "fan in" configurations. (Neither Heidemann's framework nor the interposition model suffers from this problem.) The mechanism also causes ambiguity between a given vnode pointer's role in naming the vnode it points at and in naming the stack that vnode is in. This ambiguity is the fundamental flaw of

Rosenthal's stacking model and attempts to resolve it led to our interposition model.

8. Conclusion

The interposition model has been the key to solving many of the problems that bedeviled Rosenthal's original vnode stacking model. While developing our prototype implementation we have demonstrated that:

- The interposition model works.

We have implemented a robust prototype that heavily exploits interposition yet exhibits reasonable performance. We see no significant technical barriers to bringing our prototype to production quality.

- Interposers are a powerful file system implementation and packaging tool.

We have used interposers to centralize many file system services, such as mounting, lookup performance caching, and access to special files. We expect to extend this centralization further, to directory services and media allocation policies. By doing so, we have simplified maintenance of these services and simplified the remaining file system implementations. We have made it possible to shift effort in file system development toward providing unique added value, assembling the remaining behavior from standard components.

Many issues remain, most notably the transaction and page caching issues described above. We think that our locking techniques are (barely) sufficient to allow us to avoid solving the transaction problem, but we know we must solve the page caching problem. We are encouraged by our progress so far and feel that we will be able to resolve these problems, yielding a much improved file system infrastructure.

9. Acknowledgements

Thanks go to David Rosenthal, for originating this line of work, and to the members of the UNIX International Stackable Files Work Group, for providing the impetus to resume the work after it had lain fallow for a couple years.

10. References

- [Bershad & Pinkerton 88] Bershad, Brian N. and C. Brian Pinkerton, "Watchdogs: Extending the UNIX File System", *Proceedings of the Winter 1988 USENIX Conference*, Dallas, 1988.
- [Callaghan & Lyon 89] Callaghan, Brent and Tom Lyon, "The Automounter", *Proceedings of the Winter 1989 USENIX Conference*, San Diego, 1989.
- [Guy et al 90] Guy, Richard G., John S. Heidemann, Wal Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier, "Implementation of the Ficus Replicated File System", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, 1990.
- [Heidemann 92] Heidemann, John S., "File System Development with Stackable Layers", Technical Report, Department of Computer Science, University of California, Los Angeles, 1992.
- [Kleiman 86] Kleiman, Steven R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, 1986.
- [Minnich 93] Minnich, Ronald G., "The AutoCacher: A File Cache Which Operates at the NFS Level", *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993.
- [Moran & Lyon 93] Moran, Joe and Bob Lyon, "The Restore-o-Mounter: The File Motel Revisited", *Proceedings of the Summer 1993 USENIX Conference*, Cincinnati, 1993.
- [Pendry 89] Pendry, Jan-Simon, "Amd—an Automounter", Technical Report, Department of Computing, Imperial College, London, 1989.

- [Rosenthal 90] Rosenthal, David, S. H., "Evolving the Vnode Interface", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, 1990.
- [Rosenthal 92] Rosenthal, David S. H., "Requirements for a "Stacking" Vnode/VFS Interface", UNIX International document SF-01-92-N014, Parsippany, 1992.
- [Webber 93] Webber, Neil, "Operating System Support for Portable File System Extensions", *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993.

11. Author Information

Glenn Skinner is a Senior Staff Engineer at SunSoft. He is responsible for riding herd over the SunOS architecture and, in what little time is left over, he tries to do real work. He is discovering that, no matter how hard he tries, he can't leave the influence of his years of STREAMS development behind. His e-mail address is glenn.skinner@Eng.Sun.COM.

Thomas K. Wong is a member of the Technical Staff in the File System Group at Sunsoft. He received his S.B. degree in Computer Science from MIT. He is a member of X3B11.1 and ECMA TC15, and was the project editor for the ECMA Standard 168 (ISO/IEC DIS 13490) Volume and File Structure of Read-Only and Write-Once Compact Disc Media for Information Interchange. His e-mail address is thomas.wong@Eng.Sun.COM.

Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space

Curtis Yarvin, Richard Bukowski, and Thomas Anderson

Computer Science Division
University of California at Berkeley

Abstract

In this paper, we propose a method of reducing the latency of cross-domain remote procedure call (RPC). Traditional systems use separate address spaces to provide memory protection between separate processes, but even with a highly optimized RPC system, the cost of switching between address spaces can make cross-domain RPC's prohibitively expensive.

Our approach is to use *anonymity* instead of hardware page tables for protection. Logically independent memory segments are placed at random locations in the same address space and protection domain. With 64-bit virtual addresses, it is unlikely that a process will be able to locate any other segment by accidental or malicious memory probes; it is impossible to corrupt a segment without knowing its location. The benefit is that a cross-domain RPC need not involve a hardware context switch. Measurements of our prototype implementation show that a round-trip null RPC takes only 7.7 μ s on an Intel 486-33.

1 Introduction

A traditional function of operating systems is providing *protection domains*, or areas of memory accessible only by the process which owns them. UNIX, for example, keeps every process in an entirely separate address space; other systems use a shared address space, but have a different page protection map for each process. Either way, keeping separate processes in separate protection domains provides *safety* and *security*: protection from buggy processes which accidentally touch memory locations they don't own, and protection from malicious processes trying to read or alter the memory of other processes. Safety and security are necessary in any modern operating system.

Using virtual memory hardware to enforce protection is flexible and powerful. However, it is also expensive. Giving each process its own address space increases the amount of context-specific state, and thus the cost of context switches. Context-switch cost contributes little to system overhead on normal UNIX systems; but it sets a strong lower bound on the cost of IPC. At a minimum, two context switches are required for each round-trip interprocess communication. This can limit the feasibility of splitting logically independent, but closely cooperating, modules into separate protection domains.

Software overhead in context switching once dominated the hardware cost. But, as Anderson et al. [1991] discuss, the former has decreased with processor improvements while the latter has not. The result is that context-switch times for conventionally protected systems have remained static and large, and now pose a significant impediment to extremely fine-grained IPC.

One solution is lightweight threads which share a single protection domain. Thread switching and communication is fast, but threads are not safe or secure, and even when used in a trusted environment are vulnerable to byzantine memory-corruption bugs.

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Yarvin was supported by a California MICRO Fellowship, Bukowski by an NSF Graduate Fellowship, and Anderson by a National Science Foundation Young Investigator Award.

Our goal is to build a system that combines the protection of UNIX processes and the speed of threads, to make IPC cheap enough to for heavy use. This could allow application interactions of a much finer grain than are now feasible, and large software systems could be structured as groups of cooperating processes instead of single monolithic entities.

The key to our approach is Druschel and Peterson's observation [Druschel & Peterson 1992] that, in a very large, sparse address space, virtual address mappings can act as *capabilities* [Dennis & Van Horn 1966]. If a process knows a segment's position in its address space, accessing it is trivial; without this knowledge, access is impossible. Protection can be accomplished by restricting the knowledge of segment mappings. We refer to this approach as *anonymity*.

Anonymity was not feasible before the advent of 64-bit architectures. A 32-bit address space is small enough that every valid page in the address space can be easily found through exhaustive search. But searching a 64-bit address space in this way is nontrivial. Thus, on a 64-bit machine, it is possible to randomly map unprotected pages in a shared region and use their virtual addresses as capabilities. This provides a fast and flexible way of sharing memory.

Druschel and Peterson use their approach to pass buffers between user-level protocol layers in their x-kernel system. We believe that anonymity can be used as a general-purpose protection mechanism, providing reasonable safety and security to independent processes sharing the same address space. The protection this provides is *probabilistic*, but effective. Between processes in the same address space, a context switch is a simple matter of swapping registers and stacks, and can be performed with the same efficiency as with lightweight threads.

We have developed a simple prototype of anonymous protection; our implementation can perform a round-trip null procedure call between two protected domains in only 7.7 μ s on an Intel 486-33.

The rest of this paper discusses these topics in more detail. Section 2 shows how we can use a large address space to implement probabilistic protection; Section 3 shows how we preserve anonymity during cross-domain communication. Section 4 outlines some potential uses of anonymous protection, while Section 5 outlines some limitations of our approach. Section 6 presents performance results; Section 7 considers related work. Section 8 summarizes our experiences.

2 Implementing Anonymous Protection

Safe and secure anonymity is not difficult to implement, but requires some care.

First we need a way to assign addresses. When a segment (any piece of memory that must be mapped contiguously) is loaded, it needs a virtual address. If the segment is to remain anonymous, no process that is not explicitly given this address may be allowed to discover or derive it.

So we must select the address randomly. If the address is truly random, than no better algorithm exists to discover it than brute-force search. Unfortunately, there is no such thing as a truly random number generator. The best we can do is a cryptosystem, such as DES [Nat 1977]; if the key is a secure password or code, and the plaintext an allocation sequence number, then the resulting encrypted text will be securely random. We transform the encrypted text into a virtual address and map the segment at that position (unless it would overlap another segment, in which case we compute another address).

This ensures that the most efficient algorithm for finding other processes' data will be brute-force search: iterating through virtual space and dereferencing every page (or every segment width, if segments are larger than pages), intentionally ignoring segmentation faults until a valid page is found. (The UNIX signal handling machinery, for example, allows processes to catch and ignore segmentation fault signals; for some applications this is necessary semantics.) If a process is allowed to indefinitely ignore segmentation faults, this procedure will eventually find all segments in the address space. Security is compromised when it finds the first one; we analyze how long this will take.

Memory Size	1s Delay	1ms Delay	1 μ s Delay
16MB	24162 years	24 years	1.25 weeks
256MB	1510 years	1.5 years	13.2 hours
2GB	188 years	2.3 months	1.7 hours
16GB	23 years	8.5 days	12 minutes

Table 1: Time To Breach Anonymous Security Through Brute-Force Search

The analysis has the following parameters: V , the size of the virtual address space, M , the amount of physical memory ¹, and D , the delay a process incurs on a segmentation fault. To simplify the formula, both V and M are in units of the maximum segment size. The result of the analysis is T , the expected time for a malicious process to find a segment for which it does not have the capability.

We first define $P(n)$, the probability that n memory probes will not find a mapped segment in n tries, or in other words, after nD seconds. $P(n)$ is equal to the probability that, in a sequence of V references, all of the references to the M mapped segments are after the first n references:

$$P(n) = \frac{\binom{V-n}{M}}{\binom{V}{M}}$$

To find T , we set $P(n) = 0.50$ and solve for nD . Table 1 presents some numerically calculated values for T , as a function of the delay and the amount of mapped memory, assuming a 64-bit address space and 8KByte segments. The segment size has little effect on the results presented in Table 1.

So the processor's natural fault-handling delay is unlikely to suffice, and the system must impose an additional delay penalty. The penalty should not only put the faulting process to sleep; to keep a malicious user from using multiple processes to search the address space, the penalty must also apply to all processes forked by the process's owner. However, this would still allow groups of users to divide the penalty.

The penalty time is best set as local policy. Constraints are the desired level of security, the number of users, and the added difficulty of working with faulty programs. For an currently average system, a constant delay of 1 second would seem acceptable on all fronts. As the memory size or user base of a system increases, it may be necessary to switch to adaptive delay functions which examine recent fault history. Fault history can also be examined to report suspicious patterns to the administrator.

Thus, the security we provide is not perfect; it is only probabilistic. However, we believe that the probabilities involved are low enough to make them of little concern when compared to external security issues.

Faulty software is also a threat; a faulty program may accidentally reference data it does not own. Normally, though, a failing program will stop once it causes a segmentation fault, and the likelihood that any individual accidental misreference will be to an otherwise unknown segment is quite small. The safety we provide is again probabilistic, not guaranteed, but the probabilities involved are minimal next to the chances of damage from external sources.

¹Note that M is the amount of physical memory, not the amount of the virtual address space that is in use. When faulting in a page not in primary memory, the operating system can explicitly check the segment permissions at the time of the page fault.

3 Anonymous RPC

Running separate processes in the same protection domain will provide fast context switching; to take advantage of this, we must devise an interprocess communication mechanism that preserves anonymity.

Traditional UNIX paradigms like pipes and sockets are not easy to use for fine-grained communication; they involve considerable system and application overhead. A better scheme for our purposes is remote procedure call (RPC) [Birrell & Nelson 1984]. In RPC, a *server* process exports an interface to one of its procedures; any *client* process can then bind to the procedure as if it was linked directly into the client. Local RPC has been extensively studied [Bershad et al. 1990, Bershad et al. 1991, Schroeder & Burrows 1990], and seems to be the most convenient communication paradigm for integrating software systems across domains.

Druschel and Peterson optimized their RPC system mainly for data throughput; we feel that this goal has been achieved, and optimize our anonymous RPC system – ‘ARPC’ – for round-trip latency.

3.1 Maintaining Anonymity During Communication

In principle, there is nothing to stop two processes running in the same address space from communicating directly via procedure calls. Unfortunately, this cannot be used as a protected communication protocol. In a normal procedure call, the caller must know the procedure’s address in the callee’s code segment; and must tell the callee its own address, to allow return. This is incompatible with anonymity. Even if code segments are write-protected and all data segregated, a malicious process can trace through code to find data.

To preserve anonymity, the path of control must flow through some *intermediary*: an entity which is itself protected, aware of the RPC binding, and able to manage control and data flow without revealing either party’s address to the other.

The most obvious intermediary is the kernel. Once a process traps to the kernel, it loses control of its execution, and can be shifted to any domain without having learned where that domain is. This is simple; unfortunately, it is slow. Kernel traps are typically an order of magnitude more expensive than procedure calls.

A better system can be devised if the host architecture supports execute-only page protection. Execute-only code lends itself well to anonymity; jumping to an execute-only entry point is anonymous for both sides. The caller knows the address of the callee’s text, but cannot damage that text or discover where the data might be. We cannot give the caller the actual entry point in the callee’s code, however; a jump into an arbitrary point in callee code might compromise data. Instead, we use an execute-only jump table, synthesized to contain the entry point as an immediate operand. The cost of anonymity is an extra jump.²

The data transfer protocol must also be modified to preserve anonymity. In a normal procedure call, the “server” uses the same stack as the “client” for local storage. It is therefore easy for either to corrupt the other by accidentally or maliciously writing into the wrong stack frame. Thus our RPC protocol must include a stack switch on call and on return. We must also clear registers which may contain data that can compromise anonymity.

Otherwise, anonymous RPC bears a close resemblance to ordinary procedure call, and can in some cases be performed with comparable efficiency.

3.2 ARPC Protocols

In this section we describe our ARPC protocols in considerable detail. A local RPC protocol can be constructed using either intermediary scheme. As an optimization, we design not one protocol but several, dividing RPC into several cases based on the level of *trust* between the client and server processes.

²Execute-only anonymity also requires an architecture on which branches are atomic; that is, they fully commit the processor to execution at the branch point. Some RISC architectures allow the processor to take a branch, execute an instruction, and then have the branch nullified by another branch in the delay slot.

If process A trusts that it is communicating with a non-malicious, but possibly buggy, process B, then process A can rely on the compiler and the RPC stub generator, instead of the kernel, to preserve its anonymity. This allows a more efficient implementation of cross-domain RPC. Malicious users can circumvent these utilities, but benign users are unlikely to do so accidentally.

3.2.1 Binding

Before any RPC calls can be performed, the client must *bind* to the server procedure. Binding is only performed once for each client, server, and procedure; thus it need not be optimized as heavily as the call sequence itself.

The server initiates the binding sequence, by registering an *entry point*; it gives the RPC manager the name and address of the procedure. Once the entry point is present, the client can *connect*, telling the RPC manager the name of the procedure it wants to access. A permission check may be imposed on the connection; if it succeeds, the RPC manager generates the binding.

At bind time, the RPC manager creates any necessary intermediaries, and reports the RPC entry (be it an execute-only jump table, a kernel trap, or a direct entry into the server) to the client. The manager also creates an execution stack for use in executing calls through the new binding. All our ARPC protocols statically allocate one stack per procedure binding. This may seem wasteful of memory, but stacks can be cached and unmapped when not in frequent use. The static approach eliminates the need for a dynamic stack allocation on every call.

3.2.2 ARPC, Mutual-Distrust

Our first protocol is for the most general case, when neither client nor server trusts the other. In this case we cannot jump directly from client to server, even anonymously; we also have to save the client's stack and return address. This must be done in the intermediary.³ The protocol is outlined in Figure 1.

```
push arguments and return address on client stack
enter intermediary
save return address
save registers
clear registers
save address of client stack
copy arguments to server stack
switch to server stack
leave intermediary to server procedure

execute server procedure

enter intermediary
copy return data to client stack
switch to client stack
restore saved registers
clear unsaved registers
leave intermediary to client return
```

Figure 1: ARPC Protocol, Mutual Distrust Between Client and Server

³If execute-only page protection is providing the anonymity, we jump to the intermediary through through an execute-only jump table.

For cross-domain procedure calls with few arguments, the dominant cost is saving, clearing, and restoring the registers. This is particularly true on modern processors with large register sets. For instance, on machines with register windows, the entire register set, not only the current window, must be saved, cleared, and restored on each call and return.

Marshaling of indirect parameters may not be necessary if the client organizes its memory properly. Since all addresses in the client are valid in the server, the client can keep data structures to be exported in a segment mapped separately from its private data, and pass pointers. Only if this is infeasible will explicit marshaling be needed. Direct parameters are marshaled on the client stack by the ordinary procedure-call protocol.

Note that in the absence of marshaling and the presence of execute-only code, no client stub is required. The client's view of the remote call can simply be a function pointer whose target address is the jump instruction in the anonymity table, and a generator-supplied header file can define its dereference as the function call. The ordinary argument-pushing semantics of a function call are exactly what we want.

3.2.3 ARPC, Server Not Malicious

The second protocol applies to a much more common case, when the server is trusted but the client is not. This might be the case, for example, in a small-kernel operating system. The protocol is in Figure 2.

```
save live registers
push arguments and return address on stack
call through intermediary to server

push address of client stack
switch to server stack
execute body of server procedure
pop address of client stack
push return data onto client stack
clear sensitive registers
return through intermediary to client procedure

restore live registers
```

Figure 2: ARPC Protocol, Server Not Malicious

Trusting the server not to be malicious has a considerable performance advantage. The client can anonymously jump (through the kernel or a jump table) directly to the server. In the simple case, the jump would be to a stub which would then marshal the client arguments onto the server stack. But we can achieve better performance by using an RPC generator to do a simple source code transformation of the server procedure so that it reads its arguments directly off the client stack; in this case, no copying is required. The RPC generator parses the server procedure, converts all argument references into client stack references, adds the instruction to save the client stack, and replaces ordinary returns with RPC returns.

This protocol is considerably more efficient than the mutual-distrust version. However, some additional work is required to make it protocol safe and secure.

Safety could be compromised if a register containing a client pointer became the initial value of a server variable. The solution is to ensure that all automatic server variables are initialized before use. A program which relies on the value of an uninitialized variable is unlikely to be correct, and most modern compilers can at least warn of such errors.

Likewise, security could be compromised by server data passing through registers back to the client. This is a more serious problem. The server must clear all sensitive registers before returning; if compiler analysis cannot identify which registers are sensitive, all registers must be cleared.

Another safety hole is the client stack. Although we allow the server to access its arguments on the client stack (to eliminate copying), we do not want the server to be able to inadvertently 'smash' the client stack. The solution is to prevent the server procedure from taking the address of any of its arguments; this can be enforced in the RPC generator.

3.2.4 ARPC, Neither Side Malicious

Finally, our third case: a protocol for cases in which both client and server are trusted to be non-malicious, in Figure 3.

```
save live registers
push arguments and return address on stack
call directly to server procedure

push address of client stack
switch to server stack
execute body of server procedure
pop address of client stack
push return data onto client stack
return directly to client procedure

restore live registers
```

Figure 3: ARPC Protocol, Neither Side Malicious

The only operation being performed here which is not part of a normal procedure call (assuming the caller-save register protocol) is the stack switch. And the protocol is safe; data may cross in registers, but a correctly-compiled program will never allow it to be addressed by a variable.

3.2.5 Service Management

Like LRPC [Bershad et al. 1990], ARPC is implemented by running the server procedure in the client thread. Any alternative would involve a slow interaction with operating system data structures. The logical semantics of RPC, however, imply a sleeping service thread which awakes to run the procedure and returns to sleep when it finishes. Reconciling these models requires some juggling.

One problem comes when the server crashes. We do not want the client thread to die with the server, because it was not the client who caused the error. Instead, the RPC call should return with an error condition.

Our solution is expensive, but not inappropriate given that such faults should be a rare condition. We check all the server's incoming intermediaries for saved stacks, and restore the client threads from those; if the server has any outgoing calls, we zero the stack in the intermediary so that the call faults when it returns.

This does not work for the mutual-trust protocol, which has no intermediary and saves the client stack address on the server stack. If separate crash recovery is required, a separate word must be reserved for the client stack. The separate-recovery model, in any case, is often not the preferred semantics for mutual-trust uses.

Synchronizing stack allocation is another problem. All of these routines assume one proviso: that only one thread of control, including past levels of recursion, is using the same RPC binding at any time. The assumption is convenient because it allows stacks to be statically allocated. If multiple threads must

use the same RPC binding, however, they will have to synchronize externally, as they would for any single-consumer resource.

Also, in systems which support threads, a simple lock is necessary at the beginning of untrusted-client entry points, to prevent malicious clients from sending multiple threads through the same binding and causing a stack collision.

4 Uses of Anonymous RPC

ARPC can perform many functions within a system. It can be installed at the user level, to provide additional safety within large applications that are externally protected by traditional protection domains; or it can be installed at the system level, as the main interprocess protection system; or it can be used only for some specialized tasks which require especially low latency. We consider each.

4.1 User-Level ARPC

Many large programs are written in languages, such as C, which do not guarantee the internal safety of memory. That is, code in one module may inadvertently corrupt the unrelated data of another, creating a bug which is hard to find in single-threaded code, and almost intractable in a multithreaded program.

This is an unpleasant possibility, but software designers accept it because of the high performance penalty a safe implementation would incur. In conventional systems, safety can only be ensured by actively checking each pointer dereference, or by placing separate modules in separate hardware protection domains. Either solution incurs a substantial performance penalty and neither is widely used.

ARPC offers a convenient medium. When neither side is malicious, the cost of an ARPC call is little greater than the cost of an ordinary procedure call. In most cases, replacing procedure calls with ARPC calls will have little impact on performance.

We can use this principle to increase internal safety in large software systems. Logically separate modules, and their heaps, can be placed in separate ARPC domains, providing strong probabilistic safety at minimal cost.

Such a transformation can be installed transparently in a C compiler and linker, providing increased safety invisibly to the programmer. It can also be used in compilers for languages that do guarantee memory safety, allowing them to maintain their safe semantics while providing performance competitive with C. Protecting small objects can waste memory: an anonymous object smaller than a page must still take up an entire page of physical memory.

4.2 ARPC as an Operating System Base

With some care, ARPC can be used as the sole process protection system in a traditional UNIX-style operating system. This would accelerate communication between user-level processes; it might, for example, allow fast, fine-grained drawing interaction between imaging software and a display manager.

It can also be used, in a more radical design, to improve the internal structure of the operating system. ARPC allows considerable kernel decomposition, even beyond the usual microkernel level of [Young et al. 1987]; all traditional system services – filesystem, communication, scheduling, memory management, and device drivers – can be performed at user level.

This is possible because, in an ARPC system, all common operating systems primitives can be executed without supervisor privilege. Context switches do not require manipulation of the virtual memory hardware, and process authentication can be performed by giving system servers separate ARPC entry points for each process. Even page tables and other hardware-accessed data structures can be mapped into the user address space and used directly by user-level servers. Memory-mapped devices can be treated the same way. The only functions that must be performed at supervisor level are system initialization and execution of privileged instructions. A very small kernel can handle the latter, verifying that requests come from genuine system servers by checking the source address of the trap.

This model has some practical deficiencies. UNIX semantics require separate address spaces for sibling processes after a `fork()`; although the extra space can be discarded upon the first `exec()`, handling it may prove a considerable design nuisance.

We also caution against using ARPC as the sole protection device in applications where local security is mission-critical. In an ARPC-based system, it seems too easy for small software errors in implementation – the accidental release of capabilities – to leave obscure security holes that could be exploited by dedicated intruders. This is true in any system that uses cryptographic capabilities – e.g. Amoeba [Mullender et al. 1990] – but especially so in ARPC, where every pointer is a capability.

4.3 Limited Uses of ARPC

To be useful, ARPC need not be the main protection device in a system. Its use can be restricted to special circumstances where speed is important, with traditional hardware domains used in all other places where protection is needed.

One such use might be network communications. Some new networks [Anderson et al. 1992] offer latency on the order of microseconds, a useful feature which traditional software architectures have difficulty exporting to the user. An ARPC solution would be to map the network device anonymously and give its address to a user-level device manager, which would in turn accept ARPC calls. Network-critical processes would have to share an address space with the device manager and the device; all other tasks could have their own spaces.

5 Some Problems and Disadvantages of ARPC

One potential problem with an ARPC-based environment is that image initialization is slower; since programs must be remapped randomly every time they are instantiated, they must be relocated on execution. This is a concern, but the latency of initialization is already great and relocation can be performed as part of copying the program into memory.

An anonymous address space may be very sparse and difficult to manage. If logical domains become as small as one page, a traditional multilevel page-table scheme will be prohibitively expensive. However, inverted page tables [IBM 1990] will work; as will a software-loaded TLB [Kane 1987] backed by a simple search scheme such as binary tree or hash table.

Tagging TLBs and virtual caches with process identifiers would decrease the cost of context switches and makes conventional protection more competitive with ARPC. Tagged TLBs and caches eliminate the need to flush state on context switch, one of the most expensive components of a traditional domain switch. However, not many architectures support tagged TLB's; even if they do, the switch must still be performed in supervisor mode.

It is also uncertain whether the imbalance of virtual and physical memory will continue. One cannot predict whether or not the industry will move to a 128-bit bus before available physical memory approaches 64 bits. It is worth noting, however, that in the past bus size has increased faster than physical memory size.

Also, ARPC precludes other uses for the 64-bit address space which may be superior. Large databases [Stonebraker & Dozier 1991] or distributed systems [Carter et al. 1992] can use most of a 64-bit space.

6 Performance Results

We implemented a test prototype of anonymity on an Intel 486-33 machine, capable of about 15 SPECint. The base operating system was Linux 0.98.4, a copylefted POSIX clone for the 386 architecture [Torvalds 1992]. The 486 is a 32-bit machine, and as such a truly functional implementation was impossible; however, we did our best to assure that practical concerns were treated as realistically as possible.

We did not convert the entire system to use a shared virtual address space; only specially-flagged relocatable executables were run in the same address space. Even on the 32-bit 486, it would have been

Protocol	Time (μ s)	Processor	MIPS	μ s/MIP
SRC RPC	454	C-VAX	2.7	1226
Mach RPC	95	R2000	10	950
LRPC	157	C-VAX	2.7	424
URPC	93	C-VAX	2.7	251
ARPC	7.7	i486-33	15	116

Table 2: Null RPC Performance Results

feasible to run all processes in the same space, but Linux is distributed largely in binary form, and finding and rebuilding source for all our utilities to make them relocatable would have been a task not worth the benefit.

Under this system we chose to test the slowest possible version of RPC; the protocol for two mutually untrusted processes, implemented without the use of execute-only code. Although the intermediary was accessed by traps, it ran in user mode.

Our time for a user-to-user round-trip null RPC in C was 7.7 microseconds. Of this, 3.4 μ s were due to the user-level traps; 2.4 μ s to segment peculiarities associated with the Linux implementation; 0.5 μ s to save, clear, and restore registers; and 1.8 μ s for overhead in the intermediate region. That overhead was 29 instructions; of them, 15 were involved in loading static parameters, and could be eliminated were we to synthesize the intermediary for each bind [Massalin & Pu 1989]. This is somewhat more overhead than we had hoped for, but we consider it acceptable.

It is difficult to find comparable performance figures for other RPC systems, since we do not know of any other optimized local RPC results on the same architecture. Instead, Table 2 compares our performance to that of four other RPC implementations running on other hardware: Mach RPC [Bershad et al. 1992], SRC RPC [Schroeder & Burrows 1990], LRPC [Bershad et al. 1990], and URPC [Bershad et al. 1991]. These are all optimized RPC implementations; commercial RPC implementations are frequently another order of magnitude slower.

7 Related Work

The most closely related work to ours is Druschel and Peterson's *fbufs*, packet buffers mapped in a shared anonymous space [Druschel & Peterson 1992]. User-level protocols communicate via fbufs to exchange packets at very high speed. Fbufs are a much more robust and conservative use of anonymity; the user-level protocol layers which use fbufs to communicate each have their own protection domain. Performance improvements are achieved by increased flexibility and efficiency in buffer management. Buffers do not need to be remapped in the middle of a transfer, nor must they be assigned to specific protocol pairs beforehand. Fbufs achieve data transfer rates near the theoretical limits of memory.

Other systems, like Pilot [Redell et al. 1980], use a single address space without separate hardware domains, and rely for protection on languages which restrict the use of pointers. Pilot was designed for installations which do not need security, like some personal computers, and thus benefits from the ability to optimize its protection mechanisms only for safety.

Wahbe et al. [1993] propose a different approach to enforcing protection. Instead of relying on hardware, or on anonymity, they enforce protection in software by installing extra instructions into the object code to prevent out-of-bounds pointer dereferences. Unlike Pilot, Wahbe et al.'s approach is language-independent because the protection is enforced at runtime, not in the compiler. The benefit is that separate processes can run in the same address space and communicate efficiently, at the expense of a slight slowdown in memory operations.

The Opal system [Chase et al. 1992] also uses a 64-bit shared address space, but each Opal task has its own protection domain; the goal is uniformity of naming. Using identical addresses to reference the same objects in all domains allows increased flexibility in sharing complex data structures. A context switch, however, still involves switching page tables. Other systems also follow this pattern [Carter et al. 1992].

8 Conclusion

Anonymous RPC exploits a simple property of memory systems: that it is impossible to address data whose address is unknown. With the advent of 64-bit machines, it is now possible to take advantage of this property, by placing segments at random locations in a very large, sparse address space. This allows efficient interprocess communication without expensive hardware domains.

We believe that ARPC is a useful technique for high-speed communication between closely coupled domains. Even if it is not used to organize an entire operating system, it can be used as to protect individual subsystems which use especially fine-grained communication. We suggest that software designers consider it as an option, and that hardware designers consider including features, such as execute-only page protection, which help its implementation.

9 Acknowledgements

We would like to thank Peter Druschel, Dave Keppel, and Ed Lazowska for their helpful comments on this paper.

References

- [Anderson et al. 1991] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108–120, April 1991.
- [Anderson et al. 1992] Anderson, T., Owicki, S., Saxe, J., and Thacker, C. High Speed Switch Scheduling for Local Area Networks. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 98–110, October 1992.
- [Bershad et al. 1990] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.
- [Bershad et al. 1991] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.
- [Bershad et al. 1992] Bershad, B., Draves, R., and Forin, A. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.
- [Birrell & Nelson 1984] Birrell, A. and Nelson, B. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [Carter et al. 1992] Carter, J. B., Cox, A. L., Johnson, D. B., and Zwaenepoel, W. Distributed Operating Systems Based on a Protected Global Address Space. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.
- [Chase et al. 1992] Chase, J., Levy, H., Baker-Harvey, M., and Lazowska, E. Opal: A Single Address Space System for 64-bit Architectures. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.
- [Dennis & Van Horn 1966] Dennis, J. B. and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

- [Druschel & Peterson 1992] Druschel, P. and Peterson, L. L. High Performance Cross-Domain Data Transfer. Technical report, Department of Computer Science, University of Arizona, 1992. Technical Report 92-11.
- [IBM 1990] IBM Corporation. *POWER Processor Architecture*, 1990.
- [Kane 1987] Kane, G. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.
- [Massalin & Pu 1989] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 191-201, December 1989.
- [Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44-54, May 1990.
- [Nat 1977] National Bureau of Standards. *The Data Encryption System*, 1977. Federal Information Processing Standards Publication 46.
- [Redell et al. 1980] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81-92, February 1980.
- [Schroeder & Burrows 1990] Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1-17, February 1990.
- [Stonebraker & Dozier 1991] Stonebraker, M. and Dozier, J. Sequoia 2000: Large Capacity Object Servers to Support Global Change Research. Technical report, Computer Science Division, University of California, Berkeley, July 1991.
- [Torvalds 1992] Torvalds, L. *Free Unix for the 386*, 1992. finger torvalds@kruuna.helsinki.fi.
- [Wahbe et al. 1993] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Low Latency RPC Via Software-Enforced Protection Domains. Technical report, Computer Science Division, University of California, Berkeley, April 1993.
- [Young et al. 1987] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63-76, November 1987.

10 Author Information

Curtis Yarvin is a first-year graduate student in the Computer Science Division at the University of California at Berkeley. He graduated from Brown University in 1992. His e-mail address is "curtis@cs.berkeley.edu".

Richard Bukowski is a first-year graduate student in Computer Science Division at the University of California at Berkeley. He graduated from Cornell University in 1992. His e-mail address is "bukowski@cs.berkeley.edu".

Thomas Anderson is an Assistant Professor in the Computer Science Division at the University of California at Berkeley. He received his A.B. in philosophy from Harvard University in 1983 and his M.S. and Ph.D. in computer science from the University of Washington in 1989 and 1991, respectively. He won an NSF Young Investigator Award in 1992, and he co-authored award papers at the 1989 SIGMETRICS Conference, the 1989 and 1991 Symposia on Operating Systems Principles, the 1992 ASPLOS Conference, and the 1993 Winter USENIX Conference. His interests include operating systems, computer architecture, multiprocessors, high speed networks, massive storage systems, and computer science education. His e-mail address is "tea@cs.berkeley.edu".

Integrating Handwriting Recognition into Unix

James Kempf

Nomadic Systems Group,

Sun Microsystems Computer Corp.

2550 Garcia Ave., Mail Stop MTV17-08

Mountain View, CA, 94043

Abstract

Many new portable computers are substituting an electronic stylus, or pen, for the mouse. While the pen can serve as a simple replacement for the mouse, it also provides an enhanced drawing capability. This capability opens up the potential for new modes of user interaction, one of which is text input through handwriting instead of keyboard entry. In this paper, the integration of handwriting recognition into the Unix¹ operation system is discussed. We begin with an examination of the current state of the art in recognition algorithms and how handwriting recognition can enhance a user interface. A standard application program interface for handwriting recognition engines (HRE API) is then presented. The HRE API is distinguished from existing PC operating system API's in that it is specifically designed for multiple handwriting recognition engines of differing technologies, rather than a single, vendor-specific engine, and it shares a relatively narrow surface area with the window system. The latter characteristic allows it to be used with existing window systems, such as X, but does not hinder migration to other window systems should they become available. The API has been implemented with a public domain recognition engine and is currently being circulated among vendors of handwriting recognition engines for comment. Finally, the paper concludes with a discussion of where handwriting recognition belongs in the current X window system architecture, and what would be needed to make handwriting an equal partner with typed keyboard input for text entry.

1. Introduction

An important new trend in portable computing is the substitution of an electronic stylus, or pen, for the mouse. In some machines, the pen serves as a simple replacement pointing device and text entry is still primarily through the keyboard. In other more radical designs, the keyboard has been eliminated entirely. Text entry on keyboardless machines is accomplished by writing on the surface of the display. As the user writes, the window system software leaves a trail of electronic ink, and the result is translated into text by a *handwriting recognition engine* (HRE) when input is complete. Handwriting recognition distinguishes the use of the electronic stylus in these new machines from older, desktop stylus use, in which the stylus was employed just for drawing and pointing and not for text entry. Although the use of handwriting for text entry evolved in response to the lack of horizontal support for a keyboard in portable computers, there are occasions where handwriting for text entry may be appropriate in desktop systems as well.

This paper explores the integration of handwriting recognition into the Unix operating system. In the next section, we briefly review the hardware characteristics of electronic tablets and pens. Section 3. discusses the characteristics of text input through handwriting recognition in comparison with keyboard entry. We also review of the current state of handwriting recognition technologies. Given the variety of handwriting recognition technologies available, an open systems approach to integration is essential. Section 4. through 8. describe an open handwriting recognition engine application program interface

1. Unix is a trademark of Unix System Laboratories.

(HRE API) written in ANSI C [Kernighan88]. The API can accommodate HRE's of various technologies, but has provision allowing technologies with distinguishing characteristics to offer extensions without requiring all HRE's to support them. The API also features only a few, very limited contact points with the window system. In contrast, the two major commercial handwriting recognition API's, Microsoft Windows for Pen Computing (MWPC) [Microsoft92] and PenPoint [Go92], very tightly couple the handwriting recognition API with the window system and with the look and feel of the graphical user interface. We conclude the paper in Section 9. with a design for integrating the HRE into the existing standard window system on Unix, the X window system [Scheifler86], and in Section 10. a brief summary.

2. Hardware Technologies for Pen Input

Tablet digitizers have existed for a number of years (see [Tappert88] for a review). There are currently three different technologies in widespread commercial use:

- Electromagnetic, in which the pen position is sensed by inductive coupling,
- Electrostatic, in which the pen position is sensed by capacitive coupling,
- Resistive, in which pressure on the tablet surface causes a change in resistivity, allowing the pen position to be sensed.

In addition, experimental acoustic and light sensing pen devices have been demonstrated. The most important advance in the last several years has been the coupling of the tablet with an LCD flat-panel display. With accompanying software, this allows the integrated tablet and display to mimic paper.

For handwriting recognition, tablet requirements are very strict. The tablet must have a resolution of 200 points per inch and a sampling rate of 100 points per second.

3. Characteristics of Handwriting Recognition Technologies

A common criticism of handwriting for text entry, especially from professional programmers, is that people can type much faster than they can write. In fact, studies of actual text input speed [Dao92] indicate that people require 10-20 words before coming up to full typing speed, whereas only 2-4 words are required to come up to full speed when writing. Once up to full speed, however, fast typers tend to type at about 70 words per minute while fast writers tend to write at 30 words per minute. Slow typers tend to be somewhat slower than slow writers, 15 as opposed to 20 words per minute. These data indicate that handwriting is clearly not appropriate for tasks involving entry of large amounts of text, such as composing a document or writing a program. However, for applications with command language interfaces, such as spreadsheets, or for editing documents, handwriting could actually be faster than the keyboard.

The advantage of handwriting is particularly noticeable when the application also requires interaction with a pointing device. In a study of expert spreadsheet operators [Dao92], switching between the keyboard and mouse required 25 time units, while no switching was required in a handwriting-only interface. Furthermore, time required for command and data entry in spreadsheet operation was also reduced in a handwriting-only interface due to the lack of lag in coming up to full speed for short sequences of words.

Handwriting recognizers can be differentiated into two general categories, depending on the constraints imposed on written input. *Block* recognizers require users to write cleanly separated, block-printed characters, without any overlap. They recognize handprinting on a character-by-character basis. Some block recognizers relax constraints on overlapping, but still require block printing. Since most people don't naturally write with well-separated block letters, the extra effort required makes text entry with block recognizers slower than natural handwriting. *Cursive* recognizers allow writing with characters connected by ligatures, which is how most people write naturally. Some cursive recognizers are restricted to lower case only, while others allow mixed case, and also handle block printing with overlapping characters. Most cursive recognizers restrict the input text to words in a dictionary however, in order to obtain higher translation accuracy. Both types of recognizers come in writer-independent and writer-dependent

forms. Writer-independent recognizers have good walk-up translation accuracy, and some can be improved by explicit training. Writer-dependent recognizers require the user to train the recognizer in order to get accurate translation.

Studies of recognition performance indicate that the accuracy of translation and recognition rate for current state of the art, commercial recognizers is acceptable, while still open for improvement. Users tend to judge the acceptability of translation accuracy based on the word accuracy of the result [Vallone91]. The word accuracy is the percentage of words correctly translated out of the total words entered. Character accuracy, or the percentage of characters correctly translated out of the total set of characters entered, is less important from the user's perspective. Commercial block recognizers can achieve about a 90% word recognition rate [Vallone91], while commercial cursive recognizers can achieve about the same on restricted dictionary sizes [Kempf92]. Most recognizers can achieve good recognition rates even on PC-class machines. On a 33 MHz 486 PC, cursive recognition requires between 90-180 milliseconds per character, which most users judge as excellent [Kempf92]. While improvements in translation accuracy are desirable, the current state of the art is still highly acceptable in exactly those kinds of limited text input applications where the underlying dynamics of handwriting versus typing indicate handwriting has an advantage.

A wide variety of pattern recognition and artificial intelligence technologies have been applied to the problem of on-line handwriting recognition (for a technology review, see [Tappert88], for a review of commercial products see [Gibbs93]). Chain codes, feature analysis, and templates have been applied to handwriting recognition for some time. More recently, neural nets have become popular for both cursive and block recognition [Martin92] [Mori92]. No commercial recognizer has yet exploited the full potential of learning, however. Since handwriting recognition is something which people are naturally good at, most people expect the computer to perform as well as, if not better than, a person. They expect that the computer will be able to learn their handwriting automatically, even if it is difficult for another person to read. The block recognizers distributed with MWPC and PenPoint, as well as most other block and cursive recognizers, allow users to explicitly train the recognizer for personal variations in their handwriting, but the training must be explicitly done. Ideally, training would occur by having the recognizer observe recognition errors, so that the recognition rate for the primary user improved over time without causing a deterioration in the untrained, walk-up rate.

The above analysis applies primarily to Western languages and the computer command languages derived from them. For East Asian languages, the large number of ideographic characters often requires that multiple keys be pressed to input a single character, rather than a single key as in Western languages. Although no comparative studies have been done, the difficulty of entering ideographic characters with a keyboard suggests that handwriting recognition may actually be a superior method of text entry, even for large volumes of text. Indeed, research into handwriting recognition for East Asian languages is somewhat more active than for Western languages, and commercial products are available targeted specifically to the East Asian market.

4. The Handwriting Recognition Engine API

To help foster the integration of handwriting recognition into the Unix software environment, an application program interface for handwriting recognition was designed. Recognition of text, gestures, and arbitrary objects is support by the interface. Arbitrary objects need to be supported by the API because some recognizers convert pen strokes into geometric figures. *Gestures* are noncharacter strokes which act like the commands that are typically on a menu or associated with a function key (copying, pasting, etc.). Pen-based graphical user interfaces (GUI's) employ gestures in place of some menu or keyboard commands. The advantage of a gesture is that a gesture can indicate in a single stroke a target for an action (by its position) and the action to be employed on the target (by its form). In contrast, a mouse or command key interface requires multiple mouse movements and button clicks to specify the target and activate the command.

4.1 Goals

The following is a list of explicit design goals for the API:

- Provide a functionally complete interface to HRE's in a wide variety of technologies,
- Allow particular HRE's to customize the API for technology-specific features,
- Fully support multiple HRE's both at the programmatic and system level,
- Minimize dependence on the window system API and completely decouple from the GUI look and feel,
- Fully support internationalization.

The first three goals address the open system nature of the API. The API must provide basic support for handwriting recognition service across a wide range of technologies and HRE products, without requiring support of technology-specific features from all recognizers. On the other hand, the design should have provision for extension if a particular technology or product can supply some additional service. Application developers and system integrators may want to supply different recognizer configurations for application-specific and locale-specific purposes. Therefore, the API should support the installation and use of multiple HRE's.

The fourth goal recognizes that, unlike MWPC and PenPoint, no one GUI or window system is likely to satisfy the wide range of Unix customers. In addition, since Unix has traditionally been the operating system of choice among a large segment of the computer science research community, a handwriting recognition API decoupled from any particular GUI may help to foster research into new ways of designing pen-based GUI's and new recognition technologies. Finally, the fifth goal addresses the potential interest in handwriting recognition among East Asian and other international users.

There are two important nongoals of the API:

- Support for recognizers which operate upon bitmaps,
- Support for signature verification.

Recognizers operating on bitmaps are typically associated with off-line, optical character recognition. The HRE API is specifically designed for on-line, stroke-based recognizers. For signature verification, there are additional issues, such as encryption, that need to be addressed in a full signature verification API. Signature verification also tends to use algorithms with different data requirements than handwriting recognition and doesn't support gesture translation.

4.2 Design of the API

The HRE API consists of three pieces:

- A set of C structures for transmitting information between the client application and the recognition engine,
- A functional interface for use by clients of handwriting recognition services,
- An internal structure and functional interface for use by HRE implementors.

Because the input required by the HRE and the translated output are somewhat complex, the set of structures for communicating with the HRE is rich. The structure interface can be broken into the following groups:

- Input structures for conveying pen, tablet, and other recognition information to the recognition engine,
- Output structures for conveying the results of recognition to the client,
- Information structures for conveying information about the HRE configuration or for changing it,

- Structures for passing information to/from the recognizer about gestures.
- Scalar constants and flags for indicating particular pen, tablet and other state; for example, if the pen tip is currently in contact with the tablet.

In this paper, the API functions involved in extension and translation and their associated structure arguments and returns are discussed in detail, the rest are reviewed. For more information on the HRE API, see [Kempf93b].

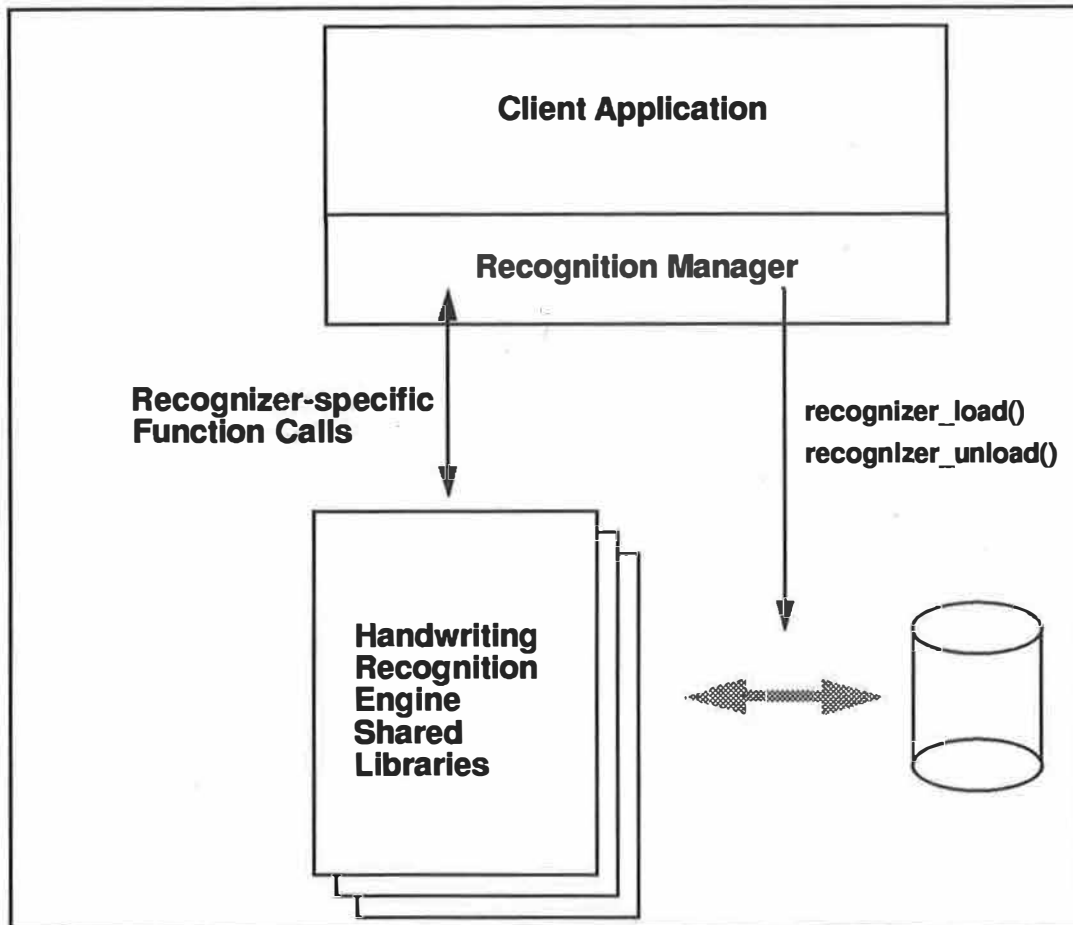


Figure 1: Architecture of the Recognition Manager

4.3 The Recognition Manager

HRE's are packaged as shared libraries [USL92a]. The HRE API implementation, or *recognition manager*, manages the interaction between the recognition client and the HRE shared library, insulating the client from differences in HRE's. The recognition manager performs the loading and unloading of recognizers, the initialization of internal recognizer state, the mapping of client-visible operation invocations into specific recognizer functions, and the finalization of the recognizer when the HRE is unloaded. The relationship between the HRE shared libraries, recognition manager, and client is illustrated in Figure 1.

The client loads a recognizer by calling the function `recognizer_load()` with arguments specifying the shared library name of a specific recognizer, any character subsets to which the recognizer should be restricted, and (optionally) the directory where the shared library is located. The opaque recognizer object returned is used as the first argument to all client API functions. The client is allowed to restrict the character sets upon which the recognizer operates because such restriction can improve translation

accuracy markedly. For example, if the client is just translating Social Security numbers, the recognizer can be restricted to the numerals. When the client is no longer in need of recognition services, the client unloads the recognizer by passing the recognizer object to the function `recognizer_unload()`.

The internal implementation of a recognizer object is a C structure with some data members and a collection of function pointers to the implementation functions in the HRE shared library, similar to a C++ virtual function table [Stroustrup91]. The structure is shown in Figure 2. In addition to the function pointers, the structure contains a starting and ending magic number (`recognizer_magic` and `recognizer_end_magic`) to check integrity, the API version number (`recognizer_version`), a pointer to a structure with the recognition configuration (`recognizer_info`), and a handle to the shared library (`recognizer_handle`).

```
typedef struct _Recognizer {
    u_int recognizer_magic;
    char* recognizer_version;
    rec_info* recognizer_info;
    void* recognizer_handle;

    ...client interface function pointers...

    u_int recognizer_end_magic;
} *recognizer;
```

Figure 2: Internal Recognizer Structure

The recognition manager calls the global function `__recognizer_internal_initialize()` in the HRE shared library after the shared library has been loaded. The shared library must implement this function to allocate a recognizer object and initialize the function pointers. It can also perform any HRE-specific initialization. The initialization function is guaranteed to be called before any client-level function. When the recognizer is unloaded, the recognition manager calls the global function `__recognizer_internal_finalize()` in the HRE shared library. This function must save any recognizer state if necessary, deallocate any internal data structures, and finally deallocate the recognizer object. The recognition manager then unloads the shared library. The recognition manager looks up the initialization and finalization functions in the shared library using the SVR4 function `dlsym(3)` [USL92b].

A recognizer may require particular internal state files for specifying the recognition characteristics of printed or written script, dictionaries of words, or other technology-specific internal state. The client API contains functions for loading and storing internal recognizer state (`recognizer_load_state()` and `recognizer_store_state()`). These two calls can be used by the recognizer to handle client requirements for user-specific or application-specific internal state, such as customized training prototypes of letters for particular users or dictionaries of words in a vocabulary specific for a particular application. In addition, the recognizer can load default prototype files when it is initialized. The API contains no functions for obtaining detailed information about letter prototypes, since this information is highly technology-specific. For example, a cursive recognizer may not be able to provide any stroke information on the words it recognizes while a block recognizer could. If a particular recognizer can supply such additional information, it can add an extension function.

4.4 Naming of Handwriting Recognizers and Data

Since recognizers are provided as shared libraries, their names must fit into the standard SVR4 scheme for naming shared libraries [USL92a], namely:

`filename.so`

where `filename` is the name of the recognition engine.

The recognition manager uses an environment variable to determine where to look for HRE's of various locales. The pathname for HRE's is:

```
$RECHOME/$LANG
```

where `RECHOME` is an environment variable set to the root directory containing the system recognizers and `LANG` is the standard SVR4 environment variable set to the current locale name [USL92c]. If `RECHOME` is not set, the directory `/usr/lib/recognizers` is searched. If `LANG` is not set, the default locale (C) is used. A client can also override these conventions by supplying a directory name argument directly to `recognizer_load()`.

An HRE may require additional files to be loaded after the shared library is loaded and before the HRE is ready to accept client requests. Any additional files should be collocated with the main HRE shared library, and either the shared library `_init()` function or the recognition engine initialization function `__recognizer_internal_initialize()` should perform the additional operations.

Some recognizers may collect information for individual users, such as individualized training prototypes. To avoid cluttering up the user's home directory with multiple files, on first use the recognition manager creates a directory for user-specific recognition data. The directory has the name `$HOME/.recognizers`, where `HOME` is environment variable containing the name of the user's home directory. By convention, an individual recognition engine puts user-specific data into files named with the HRE shared library file name, minus the `.so` extension, or, if more than one file is required, by creating a subdirectory, having the HRE name, to contain the files.

4.5 Error Handling

Client API functions return either a pointer to a translation or other data, or an integer error code. A client determines that an error has occurred if either:

- A `NULL` pointer has been returned from a function which should return a structure pointer,
- An integer error code has been returned if the function does not return a structure pointer.

Error handling in the HRE API is similar to that in dynamic linking for SVR4 [USL92b]. Each HRE implements a `recognizer_error()` function that returns a string describing the last error since the previous time `recognizer_error()` was called. The recognition manager handles its errors similarly. If no errors occurred or if `recognizer_error()` is called twice in a row without any recognition operation between, `NULL` is returned. As with the recognizers themselves, handling of error messages generated by the recognition manager is internationalized, and individual recognition engine vendors are encouraged to internationalize their error message handling as well.

5. Client Structure Interface

The client structure interface is fairly rich due to the nature of the information exchanged between the client and HRE. Clients pass information on the tablet, the physical and linguistic context in which recognition is occurring, and the actual strokes to the recognition engine. The recognition engine returns the recognized object (ASCII, variable byte, or double byte string, gesture, or arbitrary object). The recognition engine must also provide correlation between strokes and recognized text, gestures, or objects, so that a pen-based GUI can display feedback. The structure definitions reside in the file `hre.h`. Only structures directly involved in translation are discussed here, see [Kempf93b] for more information about other structures.

The structure interface uses a number of standard Unix types and a few additional scalar types. Standard Unix scalar types are taken from `sys/types.h`, while time value types taken from `sys/time.h`. Three additional scalar types are required:

- A standard boolean type is defined for boolean fields:

```
typedef u_char bool;
```

```
#define true 1
#define false 0
```

- A function type is defined for use in typing the vector of function pointers to extension functions:

```
typedef void (*rec_fn)();
```
- Since reports of recognizer confidence are restricted to the range 0 through 100, a simple type is defined for recognizer confidence:

```
typedef u_char rec_confidence;
```

5.1 Recognition Input Structures

The client uses a number of structures to pass information into the recognizer. In Figure 3, structures and constants which the recognizer must share with the window system are shown. There are only four such structures: a point structure (`pen_point`), a rectangle structure (`pen_rect`), a structure describing current pen state (`pen_state`), and a structure describing the hardware characteristics of the tablet (`tablet_info`). The `pen_point` and `pen_rect` structure definitions are identical to the definitions for the X window system [Nye92], so that rectangles and arrays of points can be passed directly to the recognizer without time-consuming copying. Because X does not yet have a standardized pen extension, the `pen_state` and `tablet_info` structures have no equivalent in X.

<pre>typedef struct { short x, y; } pen_point; typedef struct { short x,y; short width,height; } pen_rect; #define PEN_DOWN 0x1 #define PEN_BUTTON1 0x2 #define PEN_BUTTON2 0x4 #define PEN_BUTTON3 0x8 #define PEN_OUT_OF_RANGE 0x10 typedef struct { int pt_state; int pt_pressure; bool pt_invert; double pt_anglex; double pt_angley; double pt_barrelrotate; } pen_state;</pre>	<pre>#define TABLET_BARREL1 0x1 #define TABLET_BARREL2 0x2 #define TABLET_BARREL3 0x4 #define TABLET_INTEGRATED 0x8 #define TABLET_PROXIMITY 0x10 #define TABLET_RANGE 0x20 #define TABLET_RELATIVE 0x40 #define TABLET_PRESSURE 0x80 #define TABLET_HEIGHT 0x100 #define TABLET_INVERT 0x200 #define TABLET_ANGLEX 0x400 #define TABLET_ANGLEY 0x800 #define TABLET_ROTATE 0x1000 typedef struct { int ti_capabilities; u_int ti_maxx; u_int ti_maxy; u_int ti_sample_rate; u_int ti_sample_distance; } tablet_info;</pre>
---	--

Figure 3: Structure Definitions Shared with the Window System

The `ti_capabilities` member indicates basic tablet hardware capabilities, and is a combination of the `TABLET_XXX` constants using bitwise "or". The `ti_maxx` and `ti_maxy` members of the `tablet_info` structure indicate the maximum x and y coordinates that the tablet reports. All tablets are required to provide these. Some tablets may also be able to report the sampling rate, in samples/second, and the sampling distance. The sampling distance is the number of tablet coordinates the pen must move before a point event is generated. The `ti_sample_rate` and `ti_sample_distance` members indicate the sampling rate and sample distance, respectively, and are zero if a tablet is incapable of reporting the information.

The `TABLET_XXX` constants indicate the following set of capabilities (constant names for reporting in parentheses):

- A depressible pen tip or other indication that the pen is in contact with the table (required and assumed default),
- One to three barrel buttons (TABLET_BARRELx),
- Tablet is integrated with a flat panel display (TABLET_INTEGRATED),
- Tablet reports the pen location even when the pen is not in contact with the tablet. Not that this does not necessarily mean the tablet can report height information (TABLET_PROXIMITY).
- Tablet reports when the pen moves out of sensing range (TABLET_RANGE),
- Tablet only reports relative positions, like a mouse. Absolute position reporting is assumed otherwise (TABLET_RELATIVE).
- Tablet reports the pressure of the pen against the tablet (TABLET_PRESSURE),
- Tablet reports the height of the pen above the tablet (TABLET_HEIGHT),
- Tablet reports when the pen is inverted, such as would be the case during erasing with a real pen having an eraser (TABLET_INVERT),
- Tablet reports the x and y angle of the pen with the tablet surface (TABLET_ANGLEX and TABLET_ANGLEY),
- Tablet reports when the pen barrel has been rotated (TABLET_ROTATE).

The `pen_state` structure indicates the state of the various pen capabilities during a single pen stroke. The `pt_state` member is set to the logical "or" of the `PEN_xxx` constants representing the state. All tablets must report `PEN_DOWN` when the pen is in contact with the tablet. The other constants indicate additional button (`PEN_BUTTONx`) and proximity (`PEN_OUT_OF_RANGE`) state, and can be set if the corresponding flags in `ti_capabilities` member of the `tablet_info` structure indicate the hardware has that capability. Whether the rest of the `pen_state` members can be set also depends on which flags are set in `ti_capabilities`. The `pt_pressure` member is dimensionless and is positive for pressure against and negative for height above the tablet. The `pt_invert` member is `true` if the pen is inverted. The `pt_anglex`, `pt_angley`, and `pt_barrel-rotate` members are the attitude and barrel rotation angles, respectively, all in radians.

```
typedef struct {
    struct timeval ps_tstart;
    struct timeval ps_tend;
    u_int ps_npts;
    pen_point* ps_pts;
    pen_state* ps_state;
} pen_stroke;

typedef struct {
    u_short rc_upref;
    bool rc_gesture;
    rec_confidence
        rc_error_level;
    u_short rc_direction;
    u_short rc_orient;
    tablet_info* rc_tinfo;
} rc;

#define REC_LEFTH 0x1
#define REC_DEFAULT 0x0
#define REC_BOTTOM_TOP 0x1
#define REC_LEFT_RIGHT 0x2
#define REC_RIGHT_LEFT 0x3
#define REC_TOP_BOTTOM 0x4

#define REC_ULEFT 0x0
#define REC_URIGHT 0x1
#define REC_LLEFT 0x2
#define REC_LRIGHT 0x3
```

Figure 4: Stroke and Recognition Context Structures

Basic input data for the recognizer is supplied by the `pen_stroke` and `rc`, or recognition context, structures, shown along with the constants in Figure 4. The `pen_stroke` structure contains two `timeval` structure members, `ps_tstart` and `ps_tend`, for indicating the starting and ending times of the stroke. The `ps_npts` and `ps_pts` members are the number of points in the stroke and the array of points. Note that the client need not copy the array of points returned by the window system, but can simply attach a pointer on the `ps_pts` member, reducing the amount of time required to initialize the data structures for recogni-

tion. The `ps_state` member is a pointer to a `pen_state` structure, giving the state of the tablet and pen during the stroke.

The recognition context is somewhat like a graphics context in X [Nye92]. It contains particular tablet configuration information that can change but usually remains constant between calls on the recognizer. The `rc_upref` member indicates the user preferences. Currently, this member only indicates whether the user is right handed (the default) or left handed (`REC_LEFTH`). The `rc_gesture` member is set to true if the client would like the recognizer to include gestures in the translation. The `rc_error_level` member is set to the recognition confidence level, an integer in the range 0 to 100. It indicates the confidence level below which the recognizer should report that no translation could be found. The `rc_direction` member indicates the preferred and the secondary writing direction. For example, in English, the preferred direction is from left to right, the secondary direction is from top to bottom. The preferred direction is in the upper byte and the secondary writing direction in the lower byte of the `u_short`. Writing directions are indicated by the constants `REC_DEFAULT` through `REC_TOP_BOTTOM`. The `rc_orient` member gives the tablet orientation, and is set to one of the constants `REC_ULEFT` through `REC_LRIGHT`. The orientation indicates the location of the tablet origin. This information is not in the `tablet_info` structure because it can change if an application changes the tablet orientation from portrait to landscape, such as might be the case in a forms-based application. Finally, the `rc_tinfo` member contains a pointer to the underlying tablet information in a `tablet_info` structure.

```
typedef struct {
    char re_type;
    union {
        gesture* gval;
        char* aval;
        wchar_t* wval;
    } re_result;
    rec_confidence re_conf;
} rec_element;

typedef struct {
    u_int ra_nelem;
    rec_element* ra_elem;
} rec_alternative;

#define REC_NONE 0x0
#define REC_GESTURE 0x1
#define REC_ASCII 0x2
#define REC_VAR 0x4
#define REC_WCHAR 0x8
#define REC_OTHER 0x10

typedef struct Gesture {
    char* g_name;
    u_int g_nhs;
    pen_point* g_hspots;
    pen_rect g_bbox;
    void (*g_action)(
        struct Gesture*);
} gesture;

typedef void (*xgesture)(gesture*);
```

Figure 5: Basic Output Structures for Translation

5.2 Recognition Output Structures

For most recognizers, the mapping between the input strokes and output text or object is rarely one to one, so the recognizer translation functions must return arrays of alternative translations. The basic output structures for translation are shown in Figure 5. The `rec_element` structure is the basic recognition return. It contains a `re_type` field indicating the type of the returned value in the `re_result` union. Possible types are indicated by the flags on the right side of the figure. The `REC_NONE` flag indicates that no value was returned. The `REC_ASCII`, `REC_VAR`, and `REC_OTHER` flags indicate that the `aval` union member is valid. The client is responsible for casting to the appropriate type. `REC_WCHAR` indicates that the `wval` member is valid, while `REC_GESTURE` indicates that `gval` is valid. The `re_conf` member indicates the confidence (0-100, but greater than or equal to `rc_error_level`) which the recognizer places in the translation. The `rec_alternative` structure provides a way of returning a group of alternative translations for the same strokes. The translations are in the `ra_elem` array, while the `ra_nelem` member gives the size of the array.

The `gesture` structure communicates recognized gestures to the client. The `g_name` member is a distinctive gesture name. The client identifies the gesture with this name when setting the gesture action. The `g_nhs` and `g_hspots` members are the number of gesture hotspots and the hotspots themselves, respectively. Gesture hotspots are like cursor hotspots in X, and are used by the client to calculate the target of the gesture. Similarly, the `g_bbox` member, containing the bounding box around the gesture, is used to calculate where the gesture target is located. Finally, the `g_action` member is a pointer to a callback function that the client (usually a window toolkit) installs before recognition begins. Upon recognition, the client can execute the function to perform the gesture action.

```
typedef struct {
    rec_element ro_elem;
    u_int ro_nstrokes;
    pen_stroke**
        ro_stroke;
    u_int* ro_start;
    u_int* ro_stop;
} rec_correlation;

typedef struct {
    u_int rca_ncorr;
    rec_correlation* rca_corr;
} rec_corralternative;
```

Figure 6: Correlated Translation Output Structures

Information on the correlation between a translation and the original strokes is provided by the structures in Figure 6. The `rec_correlation` structure contains the `rec_element` member `ro_elem` with the translation, along with the `ro_nstrokes` and `ro_strokes` members, giving the number of strokes and a pointer to a null terminated array of pointers to strokes corresponding with the translated element. The `ro_start` and `ro_stop` members are pointers to arrays of integers giving the starting and stopping point index for the corresponding translation in each stroke. The `rec_corralternative` structure allows the return of alternative correlated translations. It contains an `rca_ncorr` member with the number of correlated translations and an `rca_corr` member with a pointer to the array of correlated translations.

```
rec_alternative**
recognizer_translate(recognizer rec,
                    rc* rec_xt,
                    u_int nstrokes,
                    pen_stroke** strokes)

rec_corralternative**
recognizer_correlate(recognizer rec,
                    rc* rec_xt,
                    u_int nstrokes,
                    pen_stroke** strokes)

int
recognizer_train(recognizer rec,
                rc* rec_xt,
                u_int nstrokes,
                pen_stroke** strokes,
                rec_element* re,
                bool replace_p)

char**
recognizer_get_gesture_names(recognizer
                             rec)

xgesture
recognizer_set_gesture_action(recognizer
                              rec,
                              char* name,
                              xgesture
                              action)

rec_fn*
recognizer_get_extension_functions(
    recognizer rec)
```

Figure 7: Translation and Extension API

6. Client Interface Functions

Figure 7 contains the translation and training functions in the API client interface. The `recognizer_translate()` and `recognizer_correlate()` functions return null terminated arrays of pointers to `rec_alternative` and `rec_corralternative` structures, respectively. These functions provide basic translation services. The `recognizer_train()` function takes a `rec_element` pointer and a boolean indicating whether the training should replace any existing translation, in addition to the translation function arguments. The `recognizer_train()` function trains the recognizer to return the `rec_element` contents when the argument strokes are recognized. The `recognizer_get_gesture_names()` function returns an array of gesture names, or NULL if no gestures are supported, and `recognizer_set_gesture_action()` sets the callback function for a particular gesture.

The `recognizer_get_extension_functions()` call returns a NULL terminated array of pointers to extension functions for the recognizer. Naturally, to use the extension functions, the recognition engine must supply interface definitions for the functions and the client must compile in the interface definitions and cast the return pointers to the appropriate type. A recognizer can also extend a structure by "laminating" members on after the standard definition [Go92]. Using a combination of extension functions and laminated structures, the HRE API achieves similar extensibility to single inheritance in C++ [Stroustrup91] (with less strict type checking), but in ANSI C. An ANSI C interface is important because the overwhelming majority of commercial HRE's, and some research ones as well, are implemented in ANSI C.

7. Recognition Examples

This section presents some examples of stroke input and the return data structures.

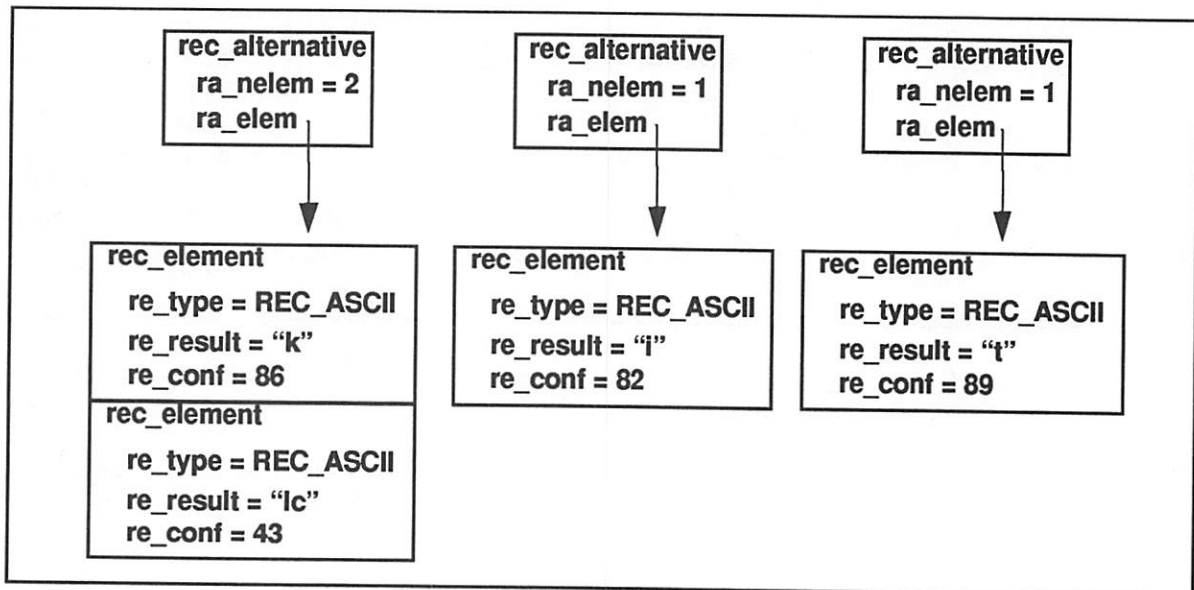


Figure 8: Returned Data Structure for Ambiguous First Letter

7.1 Ambiguous First Letter

In the first example, a block recognizer is given the following input:

k it

The first two strokes could be interpreted as either "k" or "lc".

Suppose the `recognizer_translate()` function was called with the stroke input. The returned data structure for this example would look like Figure 8. The `rec_element` structure array for the first letter contains two elements, one for the interpretation of the strokes as "k" and the other for the interpretation as "lc". The rest of the elements have only one alternative. The return value in this case would be a null-terminated array of pointers to the `rec_alternative` structures for the different letter alternatives.

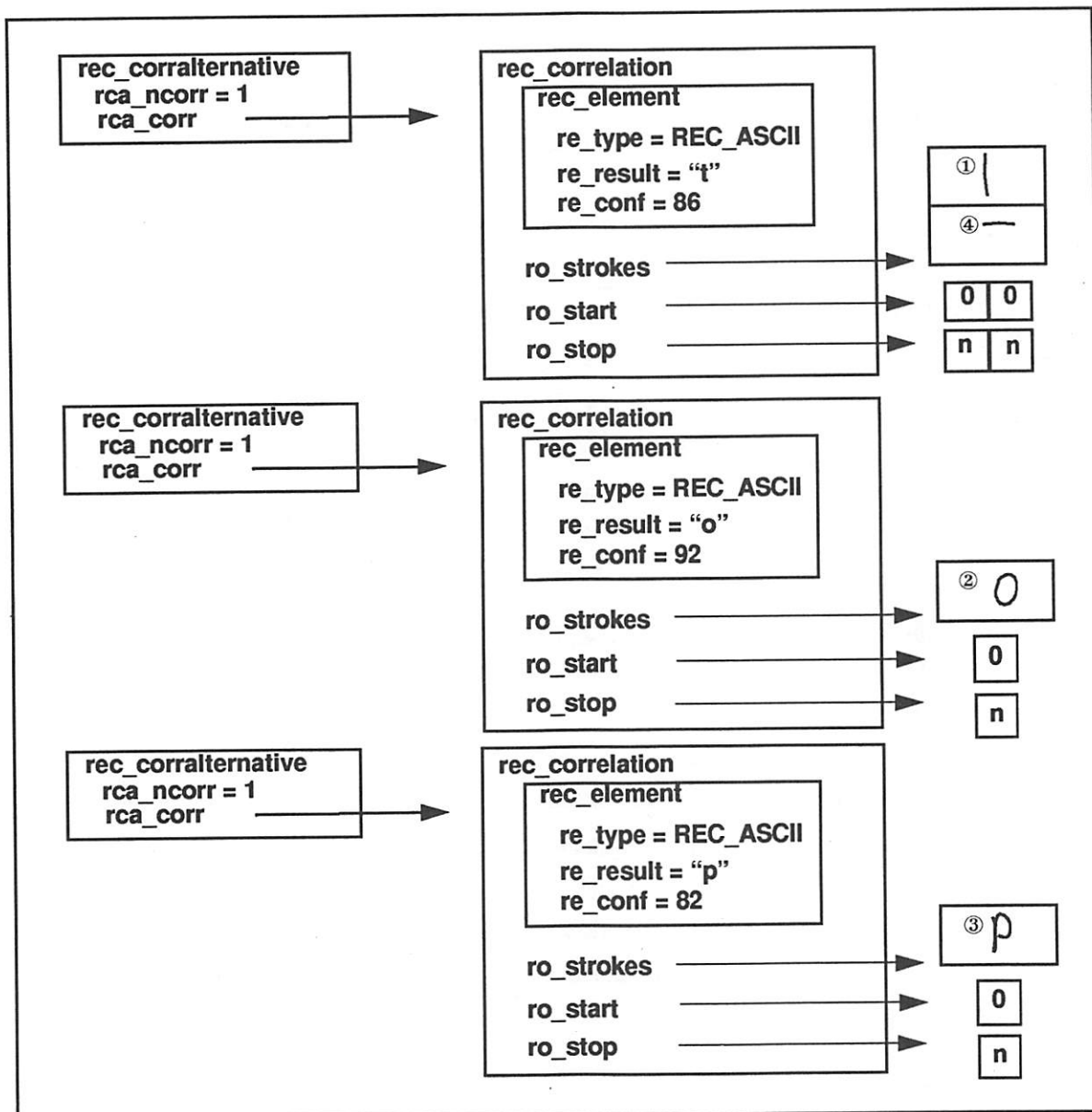
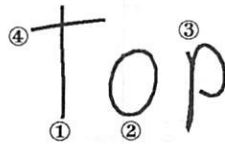


Figure 9: Stroke Correlation for Delayed Crossing of "t"

7.2 Correlation of Delayed Strokes

The next example shows how the `rec_correlation` structure can handle delayed strokes. The small numbers indicate the stroke order. The writer waited until the end of the word to go back and cross the "t":



The data structure returned from `recognizer_correlate()` is shown in Figure 9. Again, the recognizer uses a block algorithm for character-based recognition. The first correlation consists of the letter "t" with the first and fourth strokes. The other correlations are for a single stroke only.

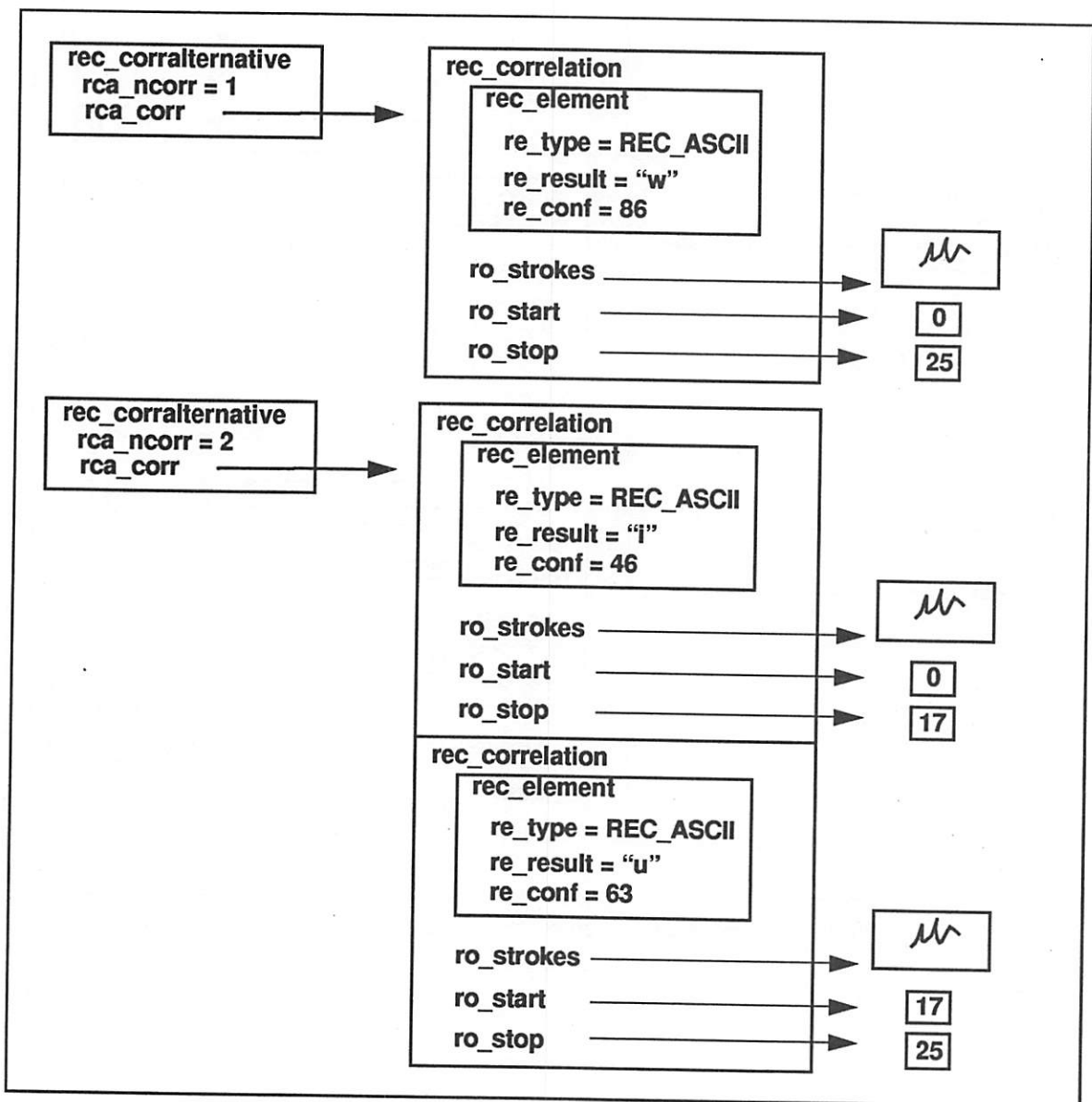


Figure 10: Multiple Translations Corresponding to a Single Stroke

7.3 Alternative Translations of a Single Stroke

The third example shows how a correlation can be established for alternative translations of a single stroke. In this case, a cursive recognizer is doing the translation, and the stroke input could be either the letter "w" or the letters "iu", with the writer having forgotten to dot the "i":



In this case, the numbers correspond to points in a single stroke.

As shown in Figure 10, the correlation alternative array has two elements, one for the translation of the stroke as "w" and the other for the translation as "iu". The start and stop arrays in the second case indicate where the recognizer stopped translating the stroke as an "i" and started translating it as "u". The low confidence levels for the second correlation indicate that the translation as "iu" is unlikely.

8. Implementation Status

The recognition manager has been implemented with a single stroke, public domain recognizer [Rubine91] as a test vehicle. To ease the task of integrating a recognizer, a collection of allocation and deallocation functions for the API data structures, somewhat like C++ constructors and destructors [Stroustrup91], were added to the recognition manager. Some of these are available to clients through the client API, since they are needed to create structures passed into the recognizer and deallocate returned structures when they are no longer needed. A GUI-based tool for testing the integration of a recognizer was built around the HRE API. It allows various recognizers to be loaded and tested through the HRE API. The API specification is currently being circulated among vendors of HRE's for comment and is available from the author upon request.

9. Handwriting Recognition and the X Window System Architecture

Since the X window system is the standard window system on most Unix platforms, a standardized API for handwriting recognition needs to be integrated into the X window system architecture. A prerequisite for handwriting recognition is a source of pen strokes. Lack of a standardized pen extension in the X11R5 distribution means that client applications wanting to employ handwriting recognition need to handle the details of pen input themselves. This implies using the primary pointer or an X input extension [Ferguson92] for the pen and implementing electronic ink by drawing lines. Requiring the client to handle the pen is particularly unattractive in a remote viewing situation, since drawing electronic ink involves a client-server round trip, which is time-consuming. An additional problem when the pen is also the primary pointing device is that, while the X input extensions allow other devices to be substituted for the mouse, the hardware characteristics must be exactly the same. As a result, it is not possible to get information specific to the pen through X, such as pressure information, when the pen is the primary pointing device.

A standardized pen extension to X would simplify handwriting recognition in applications. There have been a number of attempts to integrate pen extensions into X. An experimental pen server extension has been contributed to the X11R5 distribution by IBM [Rhyne92]. The IBM server pen extension enforces a policy of electronic inking similar to PenPoint, i.e., the pen trails ink wherever it touches the tablet. IBM has also developed two experimental Motif widgets, one for drawing and one for entering text via handwriting recognition. A simpler server extension for pen input is described in [Kemp93a]. In this extension, the server handles inking and buffers pen points until the client asks for them. Pen events are mapped into mouse events, and the server only does ink for windows which specifically register themselves as pen windows. User interface policy issues are left to the window manager and toolkit, as is typically the case in X.

Despite the lack of a standardized pen extension, design considerations suggest some alternatives for the handwriting recognition in the X architecture. Figure 11 gives an overview of how handwriting recognition could fit into a variety of X input situations. Since X applications may want to choose from a

variety of different HRE's, handwriting recognition must be available as a client library feature, and not as part of the server. Most X applications are written using a window system toolkit so the logical place to put handwriting recognition is into an extension of the toolkit. As in the IBM pen extensions, a specialized widget can handle translating pen strokes into text. The toolkit routes pen strokes to the HRE when the strokes appear on a particular widget, and displays the resulting text. Note that specialized pen widgets can also handle internationalization, as long as there is no requirement for existing text widgets to perform handwriting recognition. The input part of the X internationalization extension [Widener91] is designed to handle the translation of ASCII key codes to multibyte and wide characters. Since text input through handwriting does not involve key codes, recognition should not require an input method process. A pen extension to an existing internationalized toolkit could simply use the existing locale mechanism to determine which handwriting recognizer to load as long as text display was internationalized.

Alternatively, existing text widgets can be enhanced to support handwriting input, as is the case in MWPC, or handwriting recognition could be integrated at the level of Xlib [Nye92]. While enhancing Xlib or existing text widgets does allow current applications to transparently migrate to pen, the model of substituting pen input directly for keyboard input might not be optimal from the user's perspective. Correction of translated handwriting requires more precise hand/eye coordination for graphical input rather than the simple finger and arm movements of keyboard and mouse input. Users may perceive correcting mistranslated handwriting as requiring more concentration than simply backspacing or using the mouse to select and delete a region. Translation correction also requires some controls, such as a button to indicate when the correction should be accepted. In a study of cursive handwriting recognizers done with MWPC, users often expressed a desire for a different kind control over the translation and correction of handwriting than exists for keyboard text input windows [Kemp92]. Ultimately, a GUI "look and feel" designed around pen input, such as PenPoint offers, would provide more comfortable interaction on machines where the pen and an integrated tablet and display are the only input devices. In X, a new "look and feel" would require a new toolkit and a new window manager.

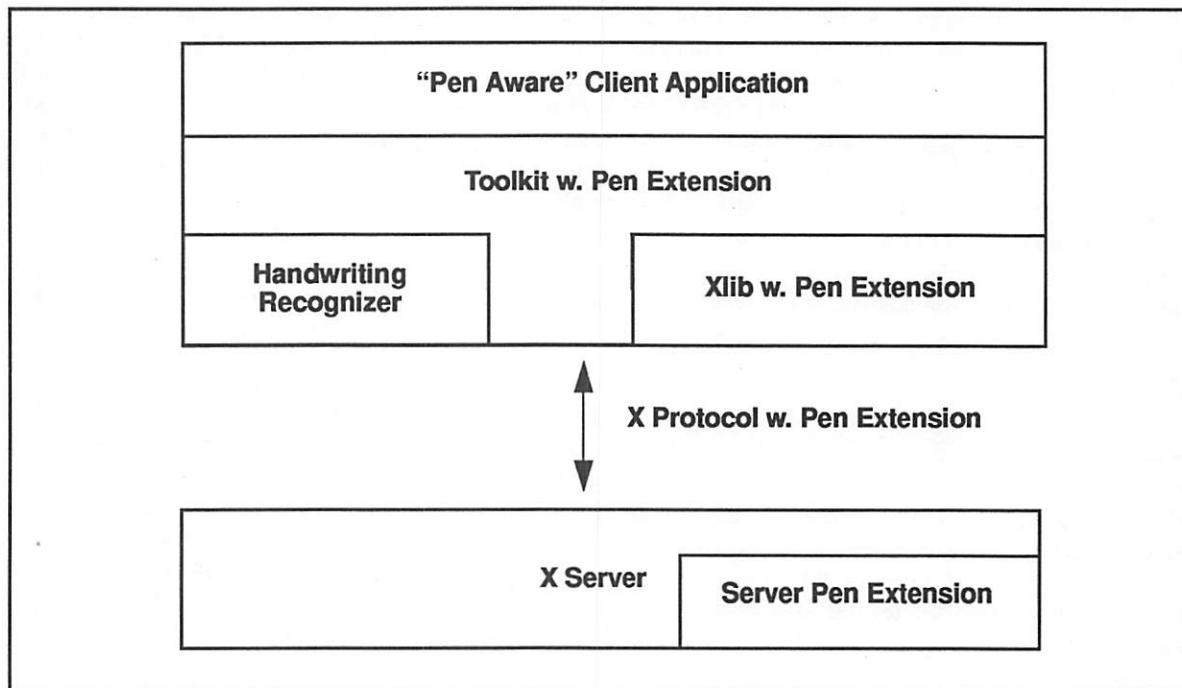


Figure 11: Integrating Handwriting Recognition into X

10. Summary

Pen input and handwriting recognition are new text input technologies developed for small, mobile PC-style computers. These technologies allow input of text without using a keyboard. The user writes directly on the display surface with an electronic stylus or pen, and the computer's operating system or window system software follows along, filling in color to provide a kind of electronic ink. The pen strokes are then translated into text by a handwriting recognition engine (HRE). Handwriting input tends to be particularly handy when a few words are needed, such as would be the case for a command language style interface or when editing a document. It is less useful when large quantities of text are required, such as when writing a document or programming. While the focus of handwriting recognition has been on mobile computers, handwriting could also play a role in desktop applications.

There are two major kinds of handwriting recognizers: block and cursive. Block recognizers require the user to print in well-separated block letters, while cursive recognizers allow letters to be connected with ligatures. Both block and cursive recognizers have a word translation accuracy of about 85-90%. Currently, a wide variety of recognition technologies are either under development or commercially available. East Asian languages look like a particularly good candidate for recognition, because the process of entering text on a keyboard in East Asian languages requires more effort.

To help foster development of applications employing handwriting recognition and research into technological improvements in recognition and handwriting-based user interfaces, a standardized HRE API was designed. The API allows multiple different recognizers to coexist and is internationalized to allow support for local language recognition. The API implementation, or recognition manager, requires the HRE to be packaged as a shared library that is dynamically loaded upon client demand. The API contains functions for translating handwriting, for obtaining translations with correlations between the translation and the original pen strokes, and for training the recognizer. The API also has a function whereby the recognizer can provide a vector of functions for extended capabilities. The recognition manager has been implemented on Solaris 2.x, Sun's distribution of SVR4.

In the current X window system environment, clients are required to deal with the details of pen input themselves, before handing off the pen strokes to the handwriting recognition engine. The alternative is to provide server and toolkit support for pen input. Two server extension prototypes and one toolkit extension prototype have already been developed. Full integration of handwriting into the X window system architecture is likely to require a standardized server extension to X for pen input, and toolkit extensions to handle handwriting recognition. Alternatively, the existing toolkit text widgets could be extended for handwriting input, though users often express a preference for a different kind of control over handwriting translation than exists for keyed text. Ultimately a very effective pen-enhanced or pen-only GUI may require a redesigned GUI "look and feel" for pen, implemented as a new toolkit and window manager on X.

11. References

- [Dao92] Dao, J., "The Next Computer Paradigm," Computer Intelligence Corp., 1992.
- [Ferguson92] Ferguson, P., "The X Input Extension," *X Resource*, 4, pp. 171-270, 1992.
- [Gibbs93] Gibbs, M., "Handwriting Recognition: A Comprehensive Comparison," *Pen*, 12, pp. 31-35, 1993.
- [Go92] *PenPoint Programmer's Reference*, Go Corp., Foster City, CA, 1992.
- [Kempf92] Kempf, J., "An Evaluation of Cursive Handwriting Recognition Technology," Sun Microsystems, 1992.
- [Kempf93a] Kempf, J., and Wilson, A., "Supporting Mobile, Pen-Based Computing with X," *X Resource*, 5, pp. 203-211, 1993.
- [Kempf93b] Kempf, J., "Preliminary Handwriting Recognition Application Program Interface for SPARC", Sun Microsystems Computer Corp., 1993.
- [Kernighan88] Kernighan, B., and Ritchie, D., *The C Programming Language: Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, 272 pp., 1988.
- [Martin92] Martin, G., and Pittman, J., "Recognizing Hand-Printed Letters and Digits," in *Advances in Neural Information Processing*, D. Touretzky, ed., Morgan Kaufman, Menlo Park, CA, pp. 415-422, 1990.
- [Microsoft92] *Microsoft Windows for Pen Computing: Programmer's Reference*, Microsoft Corp., Redmond, WA, 1992.

- [Mori92] Mori, Y., and Joe, K., "A Large-Scale Neural Network which Recognizes Handwritten Kanji," in *Advances in Neural Information Processing*, D. Touretzky, ed., Morgan Kaufman, Menlo Park, CA, pp. 415-422, 1990.
- [Nye92] Nye, A., *Xlib Programming Manual*, O'Reilly and Associates, Sebastapol, CA, 645 pp., 1992.
- [Rubine91] Rubine, D., *The Automatic Recognition of Gestures*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 1991.
- [Rhyne92] Rhyne, J., et. al., "Enhancing the X-Window System," *Dr. Dobb's Journal*, pp. 30-38, Dec., 1991.
- [Scheffler86] Scheffler, R., and Gettys, J., "The X Window System," *ACM Transactions on Graphics*, 5(2), April, 1986.
- [Stroustrup91] Stroustrup, B., *The C++ Programming Language: Second Edition*, Addison-Wesley, Reading, MA, 669 pp., 1991.
- [Tappert88] Tappert, C., et. al., "On-Line Handwriting Recognition: A Survey," *Proceedings of the 9th Conference on Pattern Recognition*, IEEE Computer Society Press, Washington, DC, pp. 1123-1132, 1988.
- [USL92a] *Unix System V Release 4: Programmer's Guide*, Unix System Laboratories, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [USL92b] *Unix System V Release 4: Interface Definition*, Unix System Laboratories, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [USL92c] *Unix System V Release 4: Multi-National Language Supplement*, Unix System Laboratories, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [Vallone91] Vallone, R., et. al., "Evaluation of Handwriting Recognition Technology: Word-level vs. Character-level Accuracy," *Proceedings of the 35th Annual Meeting of the Human Factors Society*, 1991.
- [Widener91] Widner, G., and Joloboff, V., "Developing Internationalized X Clients," *X Resource*, 0, pp. 133-152, 1991.

James Kempf (james.kempf@sun.com) has spent 10 years in the computer industry with various interesting system software projects. At Hewlett-Packard Laboratories, he worked in research on object-oriented software. Some of the projects he was involved in include participating in the design of the Common Lisp Object System (CLOS), implementing CLOS on HP Common Lisp, and developing an object-oriented database that allowed exchange of objects between programs written in Common Lisp and Objective-C. In 1988, James moved to Sun Microsystems, where he spent a year helping implement CLOS in Lucid Common Lisp and writing a prototype development environment for CLOS. In 1989, James began working on the Spring distributed, object-oriented operating system at Sun, a project which was moved to Sun Microsystems Laboratories Incorporated when it was formed in 1991. Since 1992, James has worked in Sun Microsystems Computer Corporation as a nomadic software architect and pen-based computing resource person.

Optimizing Unix Resource Scheduling for User Interaction

Steve Evans, Kevin Clarke, Dave Singleton, Bart Smaalders
SunSoft Inc.

Abstract

Techniques for improving system responsiveness for interactive end users of Unix¹ workstations are explored. After a discussion of the current state of resource scheduling, a model is presented in which dynamic input from the human user is combined with data from user interaction software to supply a centralized manager with the information necessary to determine what processes are involved with interacting with the user at any given moment. This service then communicates this process set information to the kernel, which uses it to manage memory and CPU resource allocation on the behalf of the user. Experience with a prototype of this environment is reported. An argument for an interactive scheduling class is made, along with other infrastructure changes needed to take advantage of it.

1.0 Introduction

Most implementations of the Unix operating system are not oriented to bring the power of today's workstations to the single dedicated end user's full advantage. In particular, most Unix resource management policies are oriented toward overall system throughput and fairness, not dedicated end user interactive responsiveness. In typical implementations, the portions of the kernel responsible for resource allocation cannot take into account higher level data describing user/computer interaction.

The paper describes the principal factors leading to poor interactive performance in modern Unix workstations. It outlines a user/computer interaction model that may be applied to adjust current and (near) future resource requirements in a way that favors interactivity. When this model is used to allocate system resources, a significant improvement in end user responsiveness is measured compared with traditional resource allocation algorithms.

This paper does not focus on creating a traditional real time operating environment, as has been the subject of numerous papers on deadline vs. priority scheduling, time critical computing [Khanna92], etc. Instead, the paper is about changing the operating system's focus from throughput to interactivity.

2.0 Time-sharing Today

Unix grew up trying to satisfy the needs of rooms full of university students editing, compiling programs and running troff jobs from character terminals. The technologies developed to deliver computing resources in that environment have been transferred to dedicated single user workstations. This

1. Unix is a trademark of Unix System Laboratories.

section describes how traditional Unix time-share scheduling affects end user interactivity in workstation environments.

2.1 Problem Significance

The problem of quick and predictable interactive responsiveness within Unix is significant for several reasons.

1. **Tasks Expand to Fill Available Resource** - No matter what the CPU power and available memory of a system, users, system vendors and application writers inevitably find ways to load the machine to the point at which interactivity suffers.
2. **PC User Expectations** - As Unix based and PC based operating environments are more frequently compared in the marketplace, PC users are not tolerant of user interactivity inconsistencies that more traditional Unix users can understand and explain away in terms of time-sharing notions like throughput.
3. **Human/Computer Interface Should be Smooth** - Interactive responsiveness is the most user visible aspect of a system. The user is sitting in front of the monitor seeing and experiencing responsiveness first hand. Breaks in interactivity jar the user away from the task at hand and into thinking about what is happening with the computer. The need for good interactive response is heightened by the proliferation of direct-manipulation user interfaces such as sliders, drag & drop, etc.
4. **Hungry Software** - With current resource allocation algorithms, once the available memory becomes scarce, interactive performance decreases precipitously. Demands for interactive working set for the typical windowed environment have increased significantly, exacerbating the responsiveness problem. In particular, the following innovations have contributed to the problem: increasing use of layered architectures, object-oriented languages, multiple processes providing client/server services even to the single end user, the use of inter-application messaging, multimedia, etc.
5. **Throughput is Important** - In contrast to many simpler systems, Unix workstation users expect to be able to perform many tasks simultaneously. Background compiles and links, the sending and receiving of mail, printing are all expected to run quietly in the background and not to significantly impact the "feel" or interactivity of the system. Yet at the same time, the user expects these tasks to complete quickly.

2.2 Current Practice

Current kernel memory and CPU resource control mechanisms and policies are derived from the time-sharing environment. They were born in an environment in which many terminal users were looking for good keyboard response and fair assignment of resources for background processing.

2.2.1 Processor Cycle Scheduling

Scheduling priority is the mechanism for management of CPU cycles. The scheduler factors in how much CPU time a process has used recently, how long a process has been waiting to run, a user preference as expressed by the "nice" value for the process, and other factors when determining the scheduling priority of a process. The result is that a process' scheduling priority moves like a yo-yo. Processes creep up slowly in priority as time passes when they are not allowed to run. They quickly decay in priority as they chew up cycles. There are tweaks that allow processes waiting for user input to zoom up in priority, only to decay very quickly. The kernel is actually trying to figure out which processes are interactive in nature and trying to favor them. However, it is doing this based on the assumptions germane to editing at character terminals, which are not true for many users today.

Processor scheduling is completely in the hands of the kernel. The only controls that are available to users, other than the super user, are the ability to reduce the priority of a process via `nice(1)`. This remains an effective tool for those users willing to use it. However, most users are unwilling to engage in this task. What is more, many of the processes in the system are inconvenient or unavailable for

nice(1) or renice(8) style control: daemons are owned by root, services and subprocesses are not easily located without digging around to find process ids. In addition, many processes are engaged in both interactive and batch style processing and thus have optimal nice values that change much too quickly for human management.

The only "control" available to a programmer that wants to improve interactive processing is to do very little processing at each user action. The kernel tweaks that favor such processes, mentioned above, were tuned based on the assumption that about 10,000 machine instructions would be executed by a combination of a single process and the kernel to handle a single keystroke. This tuning assumption has not changed in most Unix implementations even though the number of instructions is now many times higher, thanks to richer GUIs, and the number of processes is higher due to the multiple process architecture of the X Window System [Scheffler86].

In Unix System V, Release 4, there has been some progress in providing an architecture in which alternative scheduling approaches can be implemented. Multiple *scheduling classes* may be written by Unix vendors. To date, the default time-sharing scheduling class described above and a real-time scheduling class are available to user processes [AT&T 90]. However, the time-sharing class still lacks some of the optimizations made at Berkeley to improve interactive performance.

2.2.2 Memory Scheduling

Early Unix memory scheduling started out with swapping, executable sharing and administrator imposed memory limits as effective mechanisms for utilizing memory in a time-sharing environment. Swapping was used as the memory sharing mechanism. Executable sharing was very effective in reducing memory requirements because most of the time-sharing users were using the same small set of small programs. A fixed-size buffer cache constrained the impact of large sequential i/o operations on overall memory consumption.

Presently, many Unix systems are used as single user workstations rather than time-share systems. Here, the application mix has changed from many copies of the same small programs to many different and large programs, often infrequently accessed. The kernel has changed to rely on global page replacement as the memory sharing algorithm, with swapping used for demand reduction. There are now additional demands on memory in the form of page caches and shared memory pages[Gingell87], and the memory-based filesystem, tmpfs[Gingell87].

In SVR4, memory allocation policy is not tied to process scheduling priority. The page replacement mechanism attempts to maintain a list of free pages that are immediately available for use. Unfortunately, there is no concept of targeting non-interactive process pages when building this free list nor is there the notion of reserving the pages on the free list for interactive process demands. Memory is allocated on demand equally to all classes of processes. This has made it easier for unusually large memory demands by a single process to dramatically reduce overall performance.

2.3 Barriers to Improvement

There are practical concerns and philosophical reasons why more attention hasn't been focused on problems of interactive responsiveness. The practical concerns include:

1. **Time-sharing is Useful** - Even though systems appear to be dedicated to a single user, networked Unix boxes are actively engaged in interactions with other machines. It is very reasonable to have time-sharing oriented resource policies as the default.
2. **Prevention of Resource High-jacking** - Even on a single user machine, it is bad for overall system responsiveness if user processes are able to easily say, "make me more important". Good play-by-the-conventions applications suffer at the expense of the self centered ones. As abuses get out of hand, soon all application writers are forced to break the conventions and the situation is back to getting the user directly involved with complicated resource management.

3. **Insulating the Kernel from User Level Abuse** - The kernel has to protect itself and the machine from abuse by intentional or inadvertent resource hogs. If too much rope is given user processes, then effective use of the machine can be compromised; fair, but jerky, is better than having important programs being starved for resources to the point of corrupted functionality. Kernel folks also have to protect themselves from chasing "bug" reports about poor system performance brought about by abusive user programs.

The philosophical reasons include:

1. **Insulate the Programmer** - There has been a theme running through Unix that the programmer should not need to be involved with details of resource management. For example, C library support for grouping of dynamic memory allocations together in memory is not part of the definition of Unix. If it were, programmers would have a simple but powerful tool with which to help themselves improve localization of memory references that could lead to a dramatic reduction in working set for many programs.
2. **Don't Tie the Kernel's Hands** - The more influence that user processes have over kernel resource management the more constrained the kernel is when improving its own internal algorithms. The actual problem today is that the kernel simply does not have enough information with which to adequately improve these algorithms.
3. **It Isn't Broken** - Some Unix programmer don't consider the current situation to be broken. They don't have interactive benchmarks for which they have to find ways of improving results. They run with fast, large memory machines and are smart enough to nice compiles. Engineers must ensure that they test interactivity on low end machines in order to satisfy that important portion of the customer base using those machines.

3.0 Interactive Scheduling

A solution to improving system responsiveness should:

1. Dynamically identify processes that are involved with user interactivity.
2. Favor CPU scheduling of interactive processes.
3. Favor memory allocation towards interactive processes.

There is simply not enough information in the kernel upon which to base a predictable and understandable algorithm. Any kernel-only solution that demonstrated some success along these lines would probably incorporate too much knowledge about what is going on with user processes. The solution would not generate a mechanism that would be generally applicable as user level architecture evolved with time.

As a result, a direct approach is described in this paper in which:

1. There is a user process agent that is cognizant of user actions and can translate that behavior into user intentions.
2. Scheduling hints appropriate to carry out those intentions are formed and communicated through new kernel interfaces.
3. The kernel is augmented to respond to this new information.

3.1 The User Model

Central to identifying interactive processes is the abstract notion of *user modelling*. User modelling encapsulates the best information available about what the user has done, is doing and may be about to do, and combines that information with data from processes that are responsible for interacting with the

user. The more relevant information fed into the user model, the greater its ability to identify interactive processes. The model can be subdivided along the following lines:

1. **User Behavior and History** - Collection of information from the user.
2. **Programmatic Input** - Collection of information from programs.
3. **Interactive Process Identification** - Digestion of user focus input and programmatic input to form interactive process set identification. The point in the system responsible for this job is the *user interaction manager*.
4. **Resource Scheduling** - Allocation of resources to interactive processes by the kernel based upon user interaction manager hints.

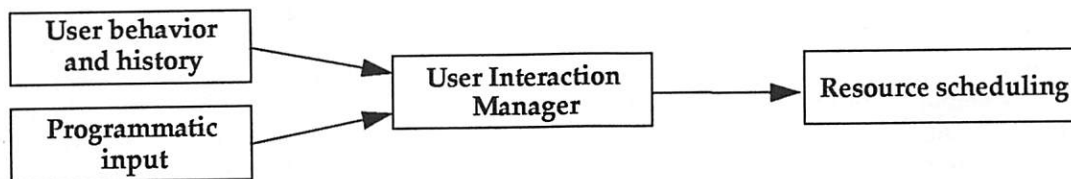


FIGURE 1. User Model Control Flow

3.2 User Behavior and History

There are two general kinds of interactivity information that can be collected from the user:

1. **Focus** - This information is collected from the user as human interactions take place with the system. The information communicates what the user is most likely focusing upon. Input device focus, e.g., keyboard and pointer focuses, if different, are the most important pieces of data. Tracking of mouse trajectory can be used to anticipate focus changes. Overall user activity monitoring can be used to detect if the user is even in front of the display.
2. **History** - Historical information can be collected for predictive purposes. Users have interaction patterns that can be exploited in terms of pre-allocating resources. The system can notice what gets used first after a period of inactivity and making sure that it has the resources necessary to run instantly. Noticing common pairs of drag sources and drop targets would allow early identification of the drop target process as interactive. To overcome application start-up wait time, the system could pre-load the application that the user is likely to use next based on historical patterns.

Also, the user could supply the system with tuning information about how to behave when trying to meet interactive needs. One could imagine a knob that ranged from "just like today" to "I value interactivity above all else" with respect to time-sharing vs. interactivity. Another knob might be used to balance a user's preference for interactivity vs. other resource demands.

3.3 Programmatic Input

Running programs supply information about which processes are interactive candidates. Running programs may also supply hints about their interactive status. More specifically:

1. **Process Identification** - The user interaction manager needs to be dynamically updated about the makeup of interactive process sets. These sets need to be associated with windows on the screen, which in turn are used as the granularity of focus control. In addition, certain processes are always required to maintain satisfactory interactive response. Examples include the X server and the win-

dow manager. In some cases the processes interacting with the user will change while the user interacts with a single window; running `vi(1)` in a terminal emulator window is a good example.

2. **Application Hints** - Programs can be the source of valuable interactivity state information. For example, noting that an application is no longer visible to the user or has become iconic may be reason enough to temporarily strip a program of its interactive state. Applications may also report their expected resource utilization patterns; `madvise(3)` is a good example.

3.4 User Interaction Manager

The user interaction manager is responsible for analysis of user behavior and programmatic input to form interactive process sets. It should have the following properties:

1. **Centralized** - The purpose of the user interaction manager is to form a model of what the user is doing, associate those actions with system components and communicate recommendations to the kernel. If the information is not centralized, it is not likely that a complete picture of the user can be correctly constructed. Thus, a model of a user interaction manager per CPU on a multiple processor host is not viable. Another reason that having the user interaction manager centralized is that one might want to move beyond a binary indication of interactive and non-interactive. One could imagine having differing interactive priorities based upon some sort of LRU algorithm or some other weighting scheme.
2. **Secure** - The user interaction manager should be thought of as a system daemon, running as root, that is trusted to make reasonable requests of the kernel, even for processes not necessarily owned by the user being modelled. One of the main reasons that it is not completely baked into the kernel is due to the importance of using window system level information.¹
3. **Efficient** - The user interaction manager had better be efficient in terms of its own CPU and memory requirements or it is possible that it could make interactivity worse instead of better.
4. **Distributed** - Even though the user model needs to be centralized, there is a control portion of the user model manager that should be distributed. For example, in the situation where an application is running on another host from where the user is interacting with it via the X Window System, the user interaction manager should be able to make interactive requests to the host running the application, on the behalf of the user, when the application has the user's focus.

3.5 Resource Scheduling

Interactive process information supplied to the kernel allows it to discard or recycle resources, concentrate current resources and pre-load soon-to-be needed resources. Resources involved include CPU cycles and memory. The kernel's responsibilities include:

1. **Favoring Interactivity** - The only real requirement is that the kernel favor interactive processes over time-sharing processes; how it does so is its business.
2. **Interactive Process Identification** - As a practical matter, part of the user interaction manager's functionality may be best implemented inside of the kernel, especially for "legacy" situations where the kernel actually has enough information to detect interactive processes on its own. For example, in the prototype, the kernel transfers interactivity to whatever process is in the terminal's controlling process group.
3. **Starvation Handling** - While maintaining interactive response is important, other tasks need to complete as well. Care must be taken that CPU intensive interactive processes do not prevent eventual completion of other important background tasks such as receiving mail, spooling print jobs, etc.

1. SunWindows, a kernel based window system, provided a simplified version of user interaction management back in 1984.

4.0 Experience

A prototype exists that incorporates enough of the aspects of the ideal solution discussed above to demonstrate the overall concept. Components of Solaris¹ 2.2 are augmented for the prototype, including the GUI toolkit, the window manager and the kernel.

4.1 User Focus Input

In the prototype, keyboard and pointer focus transitions are tracked by the window manager, which also serves as the user interaction manager. For systems using "focus-follows-mouse", the keyboard and pointer focuses are the same. For systems using "click-to-type", separate keyboard and pointer focuses are maintained. This is used to support using menus from another applications or the root window while the keyboard focus is left elsewhere. Thus, the mouse becomes a magic wand with which to dispense interactive fairy dust on running applications. The window manager conveys it's knowledge about which processes have gained or lost interactivity to the kernel via `ioctl(2)` calls to a pseudo device called the *interactive driver*. In addition to dynamically identified processes, the X server and window manager are always identified as interactive processes.

There are tuning parameters that control kernel resource management, but the end user is not expected to modify them in the prototype. Nothing is done to collect or exploit user interaction patterns. This is clearly an interesting area, e.g., typical application usage patterns during software development such as edit, compile, debug are completely cyclical in nature and hence could benefit greatly from anticipatory scheduling.

4.2 Identifying Interactive Processes

In the prototype, process identification information is communicated across the X Window System connection and maintained on the top level windows owned by the X client process. The GUI toolkit sets the process id of the client process. Widgets that manage subprocesses, e.g., the terminal emulator widget, add subprocess ids to the property.

The window server generates `EnterNotify` and `LeaveNotify` events for tracking the pointer, `FocusIn` and `FocusOut` events for tracking keyboard focus, `PropertyNotify` events for tracking changes to interactive process identification properties and `VisibilityNotify` events for tracking visibility of windows. The interactive manager is provided with enough information to track client status changes.

The kernel identifies interactive processes in two ways. First, the user interaction manager tells the kernel which processes to mark as interactive. This is done through the interactive driver. The user interaction manager calls the driver with the pid of the process it wishes to make interactive. The driver marks this process as interactive, boosts its priority and gives it a large nice value (-20). The user interaction manager can also make a process non-interactive which has the opposite effect, the processes priority is set artificially low and its nice value is set to the default (20).

Secondly, the prototype kernel transfers the interactive status of a session leader (typically a shell) when another process group becomes the controlling process group for the attached terminal. In this way, as various processes are started or stopped, placed in the background, etc., interactive status is always conferred on those processes that would be reacting to user input. This allows users to place jobs such as make, etc. in the background to reduce their impact on the rest of the system, or leave them in the foreground and maintain their high priority as long as the keyboard focus remains in that window. If a background process is brought back to the foreground, the kernel sets it to be an interactive processes again.

1. Solaris is a trademark of SunSoft, Inc.

Other than the foreground process management, the interactive state of a process is not inherited by children. While one can imagine cases where it would increase interactivity to have a child process inherit interactivity status, the prototype has taken a conservative approach. It is important to prevent the proliferation of interactive subprocesses that are unknown to the user interaction manager. This policy keeps the kernel out of the business of interactive process guessing and puts a greater burden on the user model infrastructure to identify interactive processes.

Although not done in the prototype, the property that contains the interactive process information is suitable for augmentation by application code with process ids that the GUI toolkit doesn't know about. For example, a calendar front end could discover the process id of its calendar service and add that process to the calendar's interactive set. As another example, the window manager could include the ToolTalk process id in its list of interactive system processes. That way, ToolTalk doesn't become a scheduling bottleneck in the flow of control between multiple processes engaged in inter-application messaging.

Another possibility for improvement in the prototype lies in augmenting the window's process information with internet addresses. Doing so would ensure that the user interaction manager wouldn't confuse a remote process for a local one. In addition, the interactive manager could someday make resource requests of the remote host.

4.3 User Interaction Manager

The user interaction manager could be a separate process that uses window server notification events, along with `XGetInputFocus` and `XQueryPointer` to provide window manager independent functionality, at some additional overhead. But, for the purposes of the prototype, the window manager is used as the implementation vehicle for the user interaction manager. The window manager is convenient because it already is:

1. **Centralized** - It already is a process that is dedicated to managing user interactions with the system.
2. **Secure** - Well, almost. Window managers can be allowed to run as root by changing the way that they fork programs so as to run them as a non-root user. For the prototype, the kernel allows non-root processes to adjust its interactivity assignments. This could also be handled by having the owner of the login terminal become the owner of an interactive device used for the duration of that session.
3. **Efficient** - The window manager is a particularly efficient place in the system to implement the user interaction manager: it knows about keyboard and pointer focus, it already watches for changes to top level window properties, it is involved with interesting interactivity related events like making windows iconic, and it caches information about the window tree layout.

4.4 Resource Scheduling

The prototype interactive kernel has been enhanced to simulate an *interactive scheduling class*. It schedules both CPU and memory together, giving precedence to interactive processes. The kernel allows all processes to equally share memory and CPU as long as a minimum amount of memory is immediately available for interactive processes. As long as this minimum amount of memory exists the user can switch between applications and faulting in of the pages associated with the application with the focus is relatively quick. Waiting for the page daemon to identify dirty pages and do disk writes to flush them is relatively expensive.

4.4.1 Interactive Process Scheduling

The prototype kernel gives interactive processes precedence by marking their interactive state in the `proc` structure, boosting their user priority, and setting their nice value to a -20. As the time-share scheduler recalculates priorities once a second, the -20 nice value outweighs the other factors and the interactive process stays at a fairly high priority [Leffler89][Bach86]. The prototype kernel extends the allowed

nice values so that the prototype may artificially set an interactive processes priority high and keep it high. The prototype kernel treats non-interactive processes fairly, within the constraints of priority recalculation, until free memory gets scarce.

As an aside, another modification to the time-share scheduler has been to make it act more like the BSD scheduler, that is, child process' priority and time usage at the time of a fork are a function of their parent's priority and time usage. This change stops processes from being able to completely consume the machine by forking an inordinate number of child processes that immediately run with a high priority and no time accounted to their quantum. This is not so much an interactive problem as a generic SVR4 problem. SVR4 treats the dispatch table as a state machine and does not preserve the relationship between parent and child processes as BSD does. Executing `for i in `find /usr -print` do echo $i done` shouldn't bring the system to its knees.

The time-share scheduler does most of the work for the interactive scheduling class. The user interaction manager and interactive driver reset priorities and nice values and let the time-share scheduler function as designed. The prototype has extended the time-share class and superimposed an interactive model on top of it, not actually implemented an interactive class scheduler.

4.4.2 Non-interactive Process Throttling

The prototype has a prejudice of memory over CPU. It doesn't matter how high a process' priority is if there is no free memory available. To prove this point, processes related to interactive use were set to run as real-time processes. They were scheduled to run within a bounded time period, but they had to wait for memory once they were on the processor. There was not a substantial boost in interactivity. The goal of the interactive kernel is to keep enough free memory around to allow an interactive process to immediately run. Finding enough free memory just at the time of need is a relatively expensive operation.

The kernel throttles a non-interactive process' CPU and memory usage when free memory falls below a system minimum. This value is tunable and is set in our prototype to be 1/16 of the amount of physical memory available to user processes. Throttling occurs only when memory is tight. This happens in two ways. First, each non-interactive process is preempted in favor of interactive processes whenever it exits a system call.

The second throttling is for memory hogs. If the non-interactive process is a memory hog, determined by looking at the number of valid translations the address space is using, it is preempted off the processor (also at system call exit), it has its priority lowered and its nice value is adjusted to force it to continually stay in a low priority range. A better approach would be to just gather some pages from the memory hogs, but the page daemon doesn't know which process owns which pages.

While memory is tight, non-interactive processes can use no more than 1/32 of the amount of physical memory available to user processes without being labelled a memory hog. A better implementation would involve a calculation based on the amount of physical memory and the number of extant processes. This calculation was not implemented due to time constraints and because the performance improvements already gained from other modifications were quite dramatic.

Doing throttling at system call exit has proved very successful for the prototype. However, in a production system, one would probably want to do the same calculation at the expiration of a scheduling quantum to catch pathological cases like mapping a large amount of memory and touching all the pages without doing any system calls.

4.4.3 Non-Interactive Process Starvation

In the prototype, there was a concern that there might be starvation of non-interactive processes with the throttling methods used. In fact, this was not noted with the set of lifelike loads and benchmarks used. No matter how low a process' priority was set, it would still run. There are just too many milliseconds

where the interactive processes are either sleeping or polling waiting for input. During these lull periods the non-interactive processes get to time-share the system, and until memory throttling was working correctly, these low priority non-interactive memory hogs could steal all of physical memory, even with the lowest possible priority. If one could demonstrate starvation problems, the distance between interactive and non-interactive nice values could be reduced until the problem effectively went away while maintaining the majority of the interactive benefit.

5.0 Performance

In this section, the performance of the prototype is measured under load and characterized by both objective measurements and subjective observation.

5.1 Characterizing Non-interactive Loads

Assuming that anything characterized as a background load would peg CPU utilization, minus i/o wait time, the factors that characterize loads are related to memory utilization:

1. **Page Demand** - Number of different pages touched per unit of CPU virtual time. This reflects memory requirements with respect to CPU load.
2. **Page Reuse** - The percentage of time a page is part of the page demand. For pages referenced once, this approaches 0, whereas pages accessed frequently approach 1. As used below, it refers to the average value for all pages accessed during the applications lifetime.

Using this characterization, extremes are typified by the following:

1. **Small CPU Bound Job** - Low page demand, high page reuse. This case is well-handled by the time-share scheduler; these processes decay very quickly to a low priority, and do not cause significant interactivity problems. Ray-tracing or extended-precision math programs are good examples.
2. **Multiple Process Jobs** - Low page demand, low page reuse. This is the case where a parent process creates many short-lived processes which perform some small task and then exit, typical of shell scripts, some make environments, etc. If the standard SVR4 time-sharing scheduler is fixed to act like the BSD scheduler with respect to charging child processing time to parent's doing a wait (2V) this is not a major problem for interactivity.
3. **Big CPU Bound Jobs** - High page demand, high page reuse. Typical examples are simulations, multimedia or large finite element programs. These type of processes do not coexist well with interactive use on small systems unless sufficient memory is available to contain the interactive processes working set.
4. **Streaming Memory Jobs** - High page demand, low page reuse. These are typical of programs performing a great deal of sequential disk i/o or creating large files in /tmp. Examples include copying large amounts of data via NFS, creating large temporary files in /tmp, and some compiler optimization passes. These are currently often pathological cases; the rapid page demand causes these processes to force all other processes out of memory. Some work has been done to reduce this [Mcvoy91], but this problem has not been solved. This can be prevented if either the offending process pages against itself or if the process is throttled to the point where the page daemon would grab a substantial amount of its pages. The latter is done in the prototype. With the appropriate virtual memory resource scheduling, streaming memory processes can run effectively without destroying interactivity.

When running performance tests, a loading mechanism is used that scales between high and low extremes for each type of load. The load is a process that can fork increasing numbers of child processes that perform increasing amounts of CPU activity while modifying more memory and exiting after a given amount of time. CPU and memory utilization are linked to increase at the same time in order to

reduce the complexity of the analysis, to better simulate real world loads and because more repeatable results are obtained.

5.2 Characterizing Interactive Activity

In order to measure interactive responsiveness, one has to pick meaningful interactive activities. The processes active in providing interactive response in X Window environments are the X server, window manager and client process, along with any subprocesses. In most cases, each process does a small amounts of CPU intensive work, touching a typically large number of pages, and then blocks on a system call. For even a fast typist, the various processes receive on the order of 1 char every 100 milliseconds. It is this blocking that makes boosting the CPU priority of processes stopped in the kernel so effective in boosting user interactivity in the face of CPU bound background jobs, and makes these processes so vulnerable to having their pages stolen in the case of memory intensive background loads described above.

The following scenarios are tested:

1. **Steady Interactive Activity** (*Typing Test*) - While the background load is running, the user is typing steadily. The load increases with time. The kernel and window server are instrumented to capture the elapsed time between keystrokes and character echoing. Character echoing should happen within 50 milliseconds to maintain the illusion of "instantaneous" action.
2. **Heavy Interactive Activity** (*Scrolling Test*) - While the background load is running at a constant rate the user types a command, `dd if=/etc/termcap bs=100`, that causes a large amount of scrolling in the terminal emulator. The test is rerun under various constant loads. A stop watch measurement of the elapsed time to complete the terminal output is made. Degradation of scrolling performance with increasing background load should be smooth and minimal.
3. **Changing Interactive Focus** (*System Test*) - While the background load is running at a constant rate, a script is run that drives a selection of GUI tools through a series of typical user interactions using synthetically generated input events. Numerous keyboard and pointer focuses change occur during the run. Elapsed time is measured. The test is run with control and prototype software to measure any user interaction management overhead.
4. **Warm Start of Interactive Activity** (*Warm Start Test*) - While the background load is running at a constant rate and the user is typing at a constant rate, the user stops for 30 seconds and then begins to type again. Elapsed time is measured for the first keystroke to appear. Response to the first character may be delayed, but overall performance should be regained almost immediately.

5.3 Test Environment

All tests were run on a SparcStation IPX with 16mb main memory and a local disk. The test machine was running Solaris 2.2. Typing latency measurements were gathered using an internal tracing facility [Bonwick in progress]. Trace points were inserted in the keyboard driver and the X window server.

5.4 Benchmark Results

Here are the results of the tests. The control system represents the state of the software before being modified for the prototype.

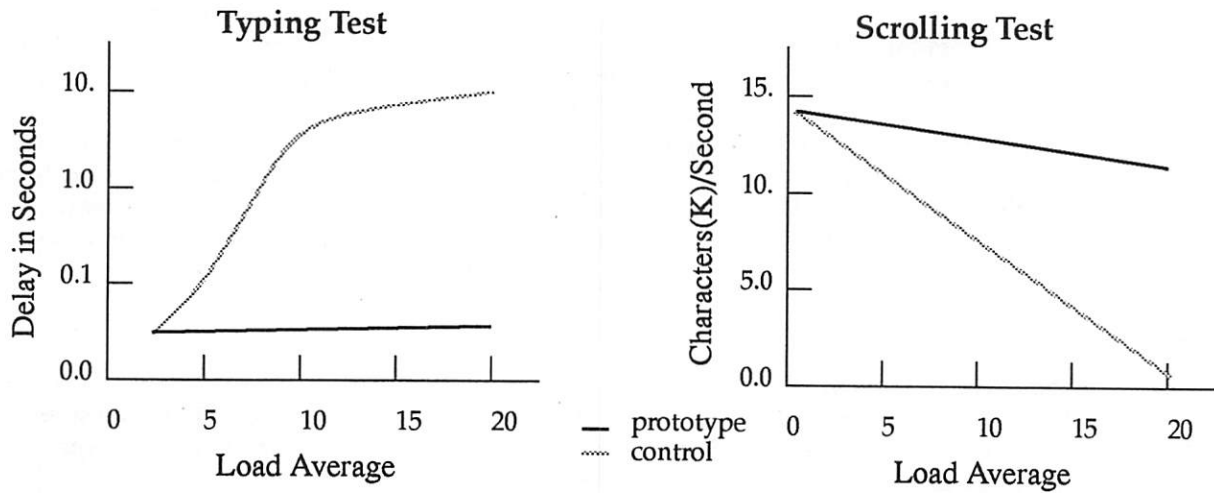


FIGURE 2. Typing and Scrolling Test Results

In Figure 2, the gray lines indicate results in the control system and the solid lines are with the prototype system. In both tests, the prototype can maintain steady interaction under various loads. If the user stops to think too long, and the memory load is high enough, this then becomes similar to the Warm Start test. The Warm Start test showed that, on the average, there was a delay of .1 seconds before the first character was echoed for the prototype system vs. 1 second for the control system.

System	No Load	Background make	Heavy Synthetic Load
Prototype	49	56	96
Control	50	89	789

TABLE 1. System Test Benchmark Results (elapsed time in seconds)

In Table 1, the System test results are shown for various loads. The prototype holds up well under the make (1) load and very well under the heavy synthetic load.

5.5 Subjective Evaluation

User of the system were generally pleased with the prototype and reported that their perception of the system was that it was faster all around; windows come up faster and things in general seem snappier. There were some interesting observations:

1. **Dueling Demos** - Two simpleminded, flat-out, non-interactive graphics demos showed uneven bursts in behavior. The window server had a tendency to completely drain the request queues before either client process had a chance to run and add anything more. When the server did wait for more

requests, a client had roughly a single scheduling quanta in which to fill the request queue. The server then drained the request queue and around it went again.

2. **Asked More of the System** - People reported being able to run a few more applications with the prototype before they started feeling paging effects similar to the non-prototype environment.
3. **Consistent Response** - Users were surprised by the levels of interactivity on heavily loaded machines. Although pathological loads can still cause problems, responsiveness during typical compilation and other software development tasks was much improved over the control system.

6.0 Future Work

Throughout the paper, mention has been made about work that was not actually tested in the prototype. Of these, there are a variety of areas of interest that are generic to multiple scheduling classes, i.e., similar issues exist for real-time and interactive scheduling classes:

1. **Page Replacement** - Global virtual memory page replacement is adequate for a machine running time-sharing. In single user workstations, some mechanism needs to be made available to apportion memory so as to maintain interactivity. The virtual memory system is going to have to utilize priority and scheduling information, in addition to LRU information, in order to provide the highest levels of interactivity. This also implies that the virtual memory system needs to become aware of which pages are used by which processes.
2. **Threads** - Scheduling in the presence of user level threads needs further analysis.
3. **Starvation** - Detection and alleviation of pathological starvation of lower priority scheduling classes by higher priority scheduling classes could bear further investigation. A *fair-share* mechanism, in which resources are allocated in pie wedges per share group, could more directly control resource allocation [Essick 90].
4. **Cooperation** - The implications of cross network cooperative scheduling is an interesting area, particularly concerning security issues.

Areas that are more specific to interactive resource management include:

1. **Broaden Programmatic Involvement** - Other conduits for programmatic interactive process notification should be explored so that services could report themselves as interactive to the user interaction manager, instead of having the user interaction manager determine process information.
2. **Refinement of User Model** - Expanding the input to the user interaction manager by exploring some of the ideas put forth in the third section should continue.
3. **Learn from Prototype** - So that all windows are registered, Xlib may be a better place in the system to stamp each window with a process id. Also, implementing the user interactive manager as an extension to the X window server would allow the user interaction manager independence from window manager choice and could make better internal scheduling decisions armed with interactive client connection data.
4. **Attitude Adjustment** - There has been considerable resistance to providing more explicit control of machine resources to application developers due to the limitations this places on the kernel. The preferred interface is for applications to report their own expected behavior to the kernel, and for the kernel to take actions as appropriate. Thus, the interactive driver interface merely communicates a process' interactive state, rather than altering priority, etc.

To promote adoption of the technologies discussed in this paper, the following should occur:

1. **Interactive Scheduling Class** - This class should actually be implemented and should follow the model provided by existing time-sharing and real-time classes.
2. **ICCCM** - A new inter-client communications convention should be adopted for the X Window System that should specify a standard for attaching interactive process ids to parts of the window tree.

These would provide a baseline set of information on which any user interaction manager could build.

7.0 Conclusions

Something between time-sharing scheduling and real-time scheduling is needed for Unix. Interactive scheduling addresses that need. The combination of an interactive scheduling class in the kernel, along with a user interaction infrastructure in user-land, suggests how it might be designed and implemented.

Modern systems have tremendous processing power. The Unix community needs to harness that power on the behalf of the end user by providing mechanisms and policies dedicated to split-second user responsiveness. The users expect it. It is up to us to deliver.

8.0 References

- [AT&T 90] "Unix System V Release 4 Internals Student Guide", Vol. I Unit 2.4.2. AT&T, 1990.
- [Bach86] Maurice J. Bach, "The Design of the Unix Operating System," Chapter 8. Process Scheduling and Time, PrenticeHall, 1986.
- [Bonwick In Progress] Bonwick, J., "Kernel Tracing in SunOS 5.0", in progress.
- [Essick 90] Essick, R., "An Event-based Fair Share Scheduler," Proceedings 1990 USENIX Winter Technical Conference.
- [Gingell87] Gingell, R., Moran, J., and Shannon, W., "Virtual Memory Architecture in SunOS," Proceedings 1987 USENIX Summer Technical Conference.
- [Khanna92] Khanna, S., Seabee, M., and Zolnowsky, J., "Realtime Scheduling in SunOS 5.0," Proceedings 1992 USENIX Winter Technical Conference.
- [Leffler89] Leffler, McKusick, Karels, Quarterman, "4.3BSD UNIX Operating System," Chapter 4.4 Process Scheduling, Addison Wesley, 1989.
- [McVoy 91] McVoy L., Kleiman, S., "Extent-like Performance from a UNIX File System," Proceedings 1991 USENIX Winter Technical Conference.
- [Scheiffler86] Scheiffler, R., and Gettys, J., "The X Window System," ACM Transactions on Graphics, 5(2), April, 1986.

9.0 Acknowledgments

The authors would especially like to thank James Hanko of Sun Microsystems Laboratories for his work on the time-share scheduler and scheduling class. Jackson Wong and Stuart Marks offered technical advice on the X server and OPEN LOOK window manager. We would also like to thank those who ran the prototype and provided valuable feedback.

10.0 Biographies

Kevin Clarke (kevin.clarke@eng.sun.com) is a Member of the Technical Staff at SunSoft, Inc., working on OpenWindows' performance issues. He is SunSoft's representative to the X Performance Characterization group.

Steve Evans (steve.evans@eng.sun.com) is a Senior Staff Engineer at SunSoft, Inc. He is currently working on architectural issues for SunSoft for the Common Open Software Environment desktop. In the past, he has worked on 2D graphics, window system components, user interface toolkits, structured graphics editing software and has done a fair bit of management.

David Singleton (dave.singleton@eng.sun.com) is a Member of the Technical Staff at SunSoft, Inc. working on OS performance.

Bart Smaalders (bart.smaalders@eng.sun.com) is a Staff Engineer at SunSoft, Inc. He is currently handling Common Open Software Environment performance issues for SunSoft. In the past, he has been the OpenWindows performance lead, multi-threaded GUI toolkits, developed distributed system administration software and built a steamboat.

AudioFile: A Network-Transparent System for Distributed Audio Applications

*Thomas M. Levergood, Andrew C. Payne,
James Gettys, G. Winfield Treese*, and Lawrence C. Stewart***

*Digital Equipment Corporation
Cambridge Research Lab*

Abstract

AudioFile is a portable, device-independent, network-transparent system for computer audio systems. Similar to the X Window System, it provides an abstract audio device interface with a simple network protocol to support a variety of audio hardware and multiple simultaneous clients. AudioFile emphasizes client handling of audio data and permits exact control of timing. This paper describes our approach to digital audio, the AudioFile protocol, the client library, the audio server, and some example client applications. It also discusses the performance of the system and our experience using standard networking protocols for audio. A source code distribution is available by anonymous FTP.

1 Introduction

Audio hardware is becoming increasingly common on desktop computers. In 1990, the authors began a project at Digital's Cambridge Research Laboratory to explore desktop audio.¹ We began by designing a flexible I/O device for audio and telephony. Once that hardware was available, we began work on software. The result of our efforts is the AudioFile System.

Similar to the X Window System [13], AudioFile was designed to allow multiple clients, to support a variety of audio hardware, and to permit transparent access through the network. Since its original implementation, AudioFile has been used for many applications and experiments with desktop audio. These applications include audio recording, playback, video teleconferencing, answering machines, voice mail, telephone control, speech recognition, and speech synthesis. AudioFile supports multiple audio data types and sample rates, from 8 KHz telephone quality through 48 KHz high-fidelity stereo.

Currently, AudioFile runs on Digital's RISC and Alpha AXP systems, on Sun's SPARC systems, and on Silicon Graphics' Indigo workstations. A source code distribution is available by anonymous FTP.

Like the X Window System, AudioFile has four main components:

- The Protocol. The AudioFile System defines a wire protocol that links the server with client applications over local and network communication channels.
- Client Library and API. The client library and applications programming interface (API) provide a means for applications to generate protocol requests and to communicate with the server using a procedural interface.
- The Server. The AudioFile server contains all code specific to individual devices and operating systems. It mediates access to audio hardware devices and exports the device-independent interface to clients.

*Also with the MIT Laboratory for Computer Science.

**The authors' names are in random order.

¹And video, but that is another story.

- Clients. The AudioFile distribution includes several out-of-the-box applications which make the system immediately usable and which serve as illustrations for more complex applications.

The parts of the implementation of AudioFile that are not specific to audio, such as client/server communications, are based on X11 Release 4.² We should emphasize that AudioFile is *not* an addition to the X Window System; it is a separate entity which borrowed some source code. We feel quite strongly that audio services should be separate from graphics.

AudioFile was designed with several goals in mind. These include:

- Network transparency. Applications can run on machines scattered throughout the network. Network transparency allows applications to run anywhere but still interact with the user. Network transparency also enables applications such as teleconferencing that need to use audio on several systems simultaneously.
- Device independence. Applications need not be rewritten to run on new audio hardware. The AudioFile System provides a common abstract interface to the real hardware, insulating applications from the messy details.
- Support for multiple simultaneous clients. Applications can run concurrently, sharing access to the actual audio hardware.³
- Support a wide range of clients. It should be possible to implement applications ranging from audio biff to multiuser teleconferencing. We chose to implement a few very general-purpose mechanisms that permit a wide variety of applications.
- Simplicity. Simple applications should be simple; complex applications should be possible.

This paper begins with a discussion of the historical context of AudioFile. We discuss the key abstractions, the network protocol, the client library, and the server implementation. Next, we describe some sample applications and analyze AudioFile's performance. We conclude with a brief discussion of our plans for future work.

2 Background

In the early 1980's, Xerox PARC built an Ethernet-based telephone system called Etherphone [15]. The system also had capabilities for workstation control of recording, playback, and storage. The Etherphone system was used primarily to explore issues of multimedia documents and computer-telephone integration.

In the mid 1980's, the Firefly multiprocessor [17] developed at Digital's Systems Research Center incorporated telephone-quality audio. An audio server buffered the past input and future output and exported a remote procedure call (RPC) interface to clients. This system pioneered explicit client control of time (as described in Section 3) and was primarily used for applications such as teleconferencing and multimedia presentations.

In the mid to late 1980's, the MIT Media Lab and Olivetti Research collaborated on a project called VOX [2]. In this system almost all audio functions were implemented inside the server, with the client merely controlling those functions. VOX was constrained by the view that clients would control the flow of audio between external devices, rather than handling the data themselves.

Other projects were underway at about the same time as AudioFile. Digital's XMedia [1] and Bellcore's Sonix [12] are similar to VOX in their emphasis on handling audio within the server. The conferencing system described by Terek and Pasquale [16] was based on a modified X server. In contrast, we think audio and graphics should be kept separate for ease of implementation and so that non-graphics equipped machines can still use audio.

3 Audio Abstractions

This section describes the fundamental abstractions of AudioFile. These provide the view of audio available to clients and guided the design of the protocol, client library, and audio server.

²Why start with a clean slate? For more information on how to steal code, consult Spencer [14].

³We think that something like an "audio window manager" might be useful, but so far we have not found it necessary to implement one.

3.1 Devices

Abstract audio devices are Analog-to-Digital (ADC) and Digital-to-Analog (DAC) converters which produce and consume sample data at a regular rate known as the sampling frequency. The sample data are one of several predefined types and consists of one or more channels.

3.2 Time

The concept of audio device time is critical to understanding the design of all AudioFile components. We expose audio device time in the protocol and at the client library API. All audio recording and playback operations in the AudioFile System are tagged with time values that are directly associated with the relevant audio hardware.

There are a remarkable number of clocks in a modern distributed computer system. A simple desktop system might have four different clocks: time-of-day, interval timer, display refresh, and audio. Each computer system in a network has its own clocks. Time-of-day clocks may be synchronized with protocols such as NTP [9], but we are not aware of any systems that keep the other clocks synchronized. In principle, it is possible to use any clock for audio. Because we wanted to be able to specify audio data down to the individual sample, we chose to use the audio sample rate clock. The server maintains a representation of this clock in a “time register” for scheduling all audio events for the particular device. AudioFile does not provide a complete infrastructure for synchronization; rather, it supplies low-level timing information to its clients. Applications can build synchronization mechanisms suitable to their own needs.⁴

Audio device time is represented by a 32-bit integer that increments once per sample period and wraps on overflow. There is no absolute reference value for a device time; the value is set to 0 when the server is initialized and advances thereafter. Time comparisons are easy to implement, as illustrated by the following code fragment for a device running at 8000 samples per second.

```
if ((b - a) > 0)      /* time b is later than time a.          */
if ((b - a) < 0)      /* time b is earlier than time a.          */
if ((b - a) == 8000) /* time b is one second later than time a.          */
```

This method breaks when the difference approaches 2^{31} . Programs that deal with time must be careful not to make comparisons between widely separated time values. Even at a 48 KHz sampling rate, however, 2^{31} samples represent about 12 hours worth of audio.

Each play and record request carries with it an exact timestamp. The abstraction is implemented by buffering future playback and recent record data in the server. Continuous recording or playback is done by advancing the requested device time by the duration of the previous request.

Explicit control of time provides the mechanism needed for real-time applications. As long as play requests reach the server before their start times, playback will be continuous. A leisurely application will schedule playback well in the future, while a real-time application will schedule for the very near future.

Recording is a much easier problem than playback. The server buffers all audio input, typically for several seconds. No data will be lost unless a request fails to reach the server until well after its start time. Because the server buffers all device input, clients can request recording at times “in the past” and deliver instantaneous response. For example, an application can begin recording at the exact moment the “Record” button is pressed, even though there is a delay from pressing the button to scheduling the record. This is a more natural interaction than requiring the user to wait for an audible beep.

This explicit use of time means that clients operate on blocks of audio data. The alternative to AudioFile’s block-oriented design would be streams, which would cause problems that AudioFile avoids. It is difficult to determine how much data is buffered in a stream or to find out if a stream is running or blocked. Streams tend to obscure issues of bandwidth and latency that are critical to real-time applications. Finally, in practice, applications deal with data in blocks anyway.

3.3 Output Model

AudioFile’s output model is shown in Figure 1. Clients can schedule playback at any time from the present to four seconds into the future. Requests that fall in the past are silently discarded. Requests that fall beyond the four-second

⁴We envision adding standard mechanisms for providing clock conversion services, but have not yet encountered a compelling need to do so.

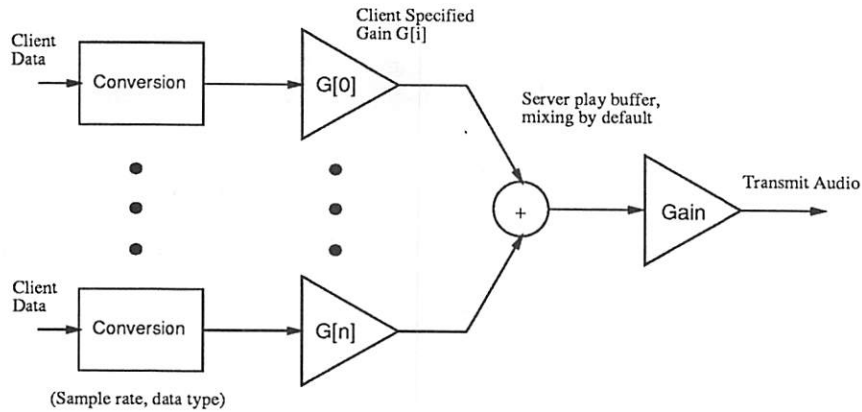


Figure 1: AudioFile server output model

buffer are suspended until time advances to within four seconds.

After a playback request is received by the server, the data are passed through an optional module that converts the client's data type to the device's preferred representation.⁵ After conversion, a client specified gain is applied, and the data are combined with the data from other clients in the playback buffer. The server will mix client data by default, but preemptive playback is possible. Finally, a master volume control is applied.

The output model specifies that silence is emitted during periods of time in which no data have been written to the output buffer. This may reduce network bandwidth requirements, since clients need not transmit silence data to the server.

3.4 Input Model

AudioFile's input model is shown in Figure 2. As in the output model, the server buffers four seconds of data. The recorded data are modified by a master input gain and placed into the server buffer. Clients requesting input data older than four seconds in the past are given silence. Requests within the past four seconds return buffered data, and requests in the future block until time advances. The system also supports a non-blocking record request, which will return as much data as possible without blocking. The input model also supports optional modules which convert the native audio hardware data type to the client requested type.

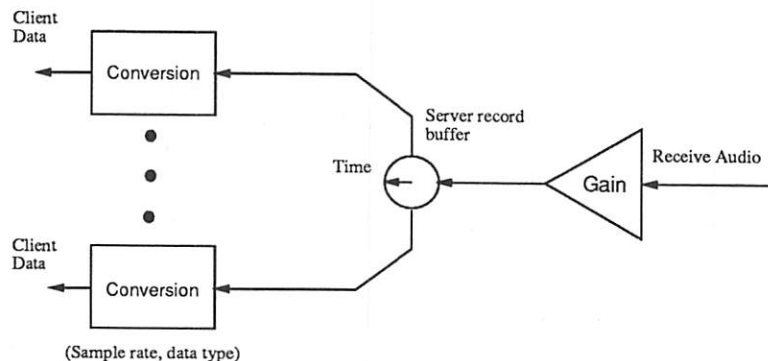


Figure 2: AudioFile server input model

⁵Conversions could include sample rate changing as well as data type conversion.

3.5 Events

Events are asynchronous messages from server to client. Events may be generated by a device or as a side effect of some client's request. Clients must register their interest to receive various classes of events, such as a telephone device ringing or a change in a property used for interclient communications.

4 Audio Hardware

AudioFile is designed to support a variety of audio hardware. Currently supported devices include:

LoFi LoFi,⁶ designed at CRL, is a workstation peripheral later released as the DECaudio product [7]. LoFi supports two 8 KHz CODECs, one connecting to a telephone line. LoFi contains a digital signal processing chip with 32K words of memory shared with the host. The DSP supports a 44.1 KHz stereo DAC and can also be used with external "DSP port" devices operating at up to 48 KHz. The telephone interface on LoFi enables applications such as voice mail and remote information access.

We see no difficulty in supporting other kinds of telephone interfaces, such as ISDN or PhoneStation [19].

JVideo JVideo is an experimental desktop video peripheral. Like LoFi, JVideo includes audio hardware based on a DSP and shared memory. However, JVideo has neither telephony capability nor an external DSP port.

Baseboard The Personal DECstation series, the Alpha AXP-based systems, and the SPARCstation 2 all include 8 KHz CODEC devices on their system modules.

Indigo The Silicon Graphics Indigo workstations support stereo audio at rates up to 48 KHz.

LineServer LineServer is an Ethernet peripheral used within Digital's research labs for remote bridging and routing. LineServer includes an 8 KHz CODEC. The LineServer version of the AudioFile server is interesting because the server runs on a nearby host, not on the LineServer itself.

5 Protocol Description

The AudioFile protocol is modeled on the same basic principles as the X Window System protocol. AudioFile can be used over any transport protocol that is reliable and which does not reorder or duplicate data. The current version supports TCP/IP and UNIX-domain sockets. At connection setup, the client and server exchange version information, the server sends audio device attributes, and clients provide authentication information.

The AudioFile protocol defines 37 request types.⁷ Many of these are for housekeeping purposes, including access control, interclient communications, and extensions (though no extensions are implemented today). Other requests support device control, telephony, and audio data handling. Table 1 summarizes AudioFile's protocol requests. All protocol requests have a length field, an opcode, and an opcode extension. Header fields are kept naturally aligned by padding request data out to a 32-bit boundary.

There are five fixed-size events supporting telephone control and interclient communications. All events contain the audio device time and the clock time of the host of the server. The host clock time may be needed when synchronizing with other media.

Rather than specifying all parameters for play and record with each request, a client uses an "audio context" (AC) to encapsulate most of these parameters. The audio context includes the play gain, number of channels, sample type, and byte order. ACs simplify the programming interfaces for play and record.

AudioFile adopted from X the idea of property lists to enable clients to communicate. Properties are associated with a device, and can be read and written by clients. Clients can register to be notified when properties change. These facilities can be used to coordinate use of resources and to share information, such as the last phone number dialed.

6 Client Libraries

We have developed two client libraries: a standard interface to the AudioFile server and a utility library of common functions required by many clients.

⁶We called it LoFi primarily because it wasn't. LoFi always included high fidelity audio capability.

⁷For comparison, the X Window System has 119 requests in the core protocol.

Audio and Events	SelectEvents CreateAC ChangeACAttributes FreeAC PlaySamples RecordSamples GetTime	Select which events the client wants Create an audio context Change the contents of an audio context Free an audio context Play samples Record samples Get the audio device's time
Telephony	QueryPhone EnablePassThrough DisablePassThrough HookSwitch FlashHook EnableGainControl DisableGainControl DialPhone	Get telephone state Enable telephone passthrough Disable telephone passthrough Control hookswitch Flash hookswitch Not for general use Not for general use Obsolete, do not use
I/O Control	SetInputGain SetOutputGain QueryInputGain QueryOutputGain EnableInput EnableOutput DisableInput DisableOutput	Set input gain Set output gain (volume) Find out current input gain Find out current output gain Enable input Enable output Disable input Disable output
Access Control	SetAccessControl ChangeHosts ListHosts	Set access control Change access control list List which hosts are permitted access
Atoms and Properties	InternAtom GetAtomName ChangeProperty DeleteProperty GetProperty ListProperties	Allocate unique ID Get name for ID Change device property Remove device property Retrieve device property List all device properties
Housekeeping	NoOperation SyncConnection QueryExtension ListExtensions KillClient	Non-blocking NoOperation Round-trip NoOperation Not yet implemented Not yet implemented Not yet implemented

Table 1: AudioFile protocol requests

6.1 Core Library

The core client library is the standard interface for AudioFile clients. Some of its functions provide interfaces to the AudioFile protocol; others provide an interface to the library's internal data structures. Table 2 summarizes these library functions.

Some library functions, such as `AFGetTime()`, require an immediate response from the server; others, such as `AFCreateAC()`, do not. In the former case, the library blocks until a reply is received. Otherwise, the library may defer sending the request and will return to the client immediately. Certain operations, including the synchronous functions, flush any deferred requests.

AudioFile provides a simple access control scheme based on host network address. The access control functions allow hosts to be added or removed from the access list and allows access control to be disabled entirely.

6.2 Client Utility Library

The AudioFile distribution also includes a utility library to provide a number of facilities that are used by several clients. Two kinds of facilities are provided: tables and subroutines.

The AudioFile system handles a variety of digital audio data formats, such as μ -law and A-law. The utility library includes tables for converting these formats to and from linear encoding. The library also includes tables for computing signal power, gain control, and generating sine waves at various frequencies.

The utility library gathers together a number of useful subroutines. Most do not directly interact with the AudioFile protocol. They include subroutines for generating on-the-fly gain translation tables for μ -law and A-law samples, tone generation procedures, and some miscellaneous functions. One subroutine, `AFDialPhone()`, encapsulates the operations necessary to generate Touch-Tone dialing sequences on a telephone device.

7 Server Design

The AudioFile server is responsible for managing the audio hardware and presenting abstract device interfaces to clients via the AudioFile protocol. This section discusses some of the important issues in the server's design, the implementation of buffering to provide the audio device abstraction, and some other details of the server's implementation.

7.1 Implementation Considerations

Our primary concern for the implementation of an AudioFile server was performance. We wanted the server to run continuously in the background, so we felt that the quiescent server should present a negligible CPU load. Further, load due to the server with a few clients running should leave most of the CPU available for applications.

We considered using threads to implement the server, but were apprehensive about the performance and portability of existing thread packages. Although the internal structure of the server might be slightly cleaner with threads, we took the safer route and designed the server as a single-threaded process. The server must handle requests from multiple active clients, so we designed it so that one client cannot dominate the available processing time.

7.2 Buffering

The server maintains input and output buffers for each audio device. A periodic update task moves samples between the server buffers and the audio hardware. Figure 3 illustrates the server record and play buffers before and after the update task executes. At each invocation, the update task moves new record data (since `recLastUpdate`) from the hardware buffer to the server buffer, and moves the next batch of playback data (starting at the "before" `timeNextUpdate`) from the server buffer to the hardware buffer. Finally, the update task initializes the end of the server buffer with silence.

Unless a client request falls into the shaded portions of Figure 3, it can be handled entirely out of the server buffers. If a record request falls after `recLastUpdate`, the server performs an update before handling the request. If a playback request falls before `timeNextUpdate`, the server writes the data all the way through to the hardware.

Connection Management	AFOpenAudioConn AFCloseAudioConn AFSynchronize AFSetAFTERFunction	Open a server connection Close a server connection Synchronize with the audio server Set a synchronization function
Audio Handling	AFPlaySamples AFRecordSamples AFGetTime	Play digital audio samples Record digital audio samples Get the device time of a device
Audio Contexts	AFCreateAC AFChangeACAttributes AFFreeAC	Create a new audio context (AC) Modify an AC Free resources associated with an AC
Event Handling	AFEventsQueued AFPending AFIfEvent AFCheckIfEvent AFPeekIfEvent AFNextEvent AFSelectEvents	Check for events Returns number of unprocessed events Find and dequeue a particular event (blocking) Find and dequeue a particular event (nonblocking) Find a particular event (blocking) Return the next unprocessed event Select events of interest
Telephone	AFCreatePhoneAC AFFlashHook AFHookSwitch AFQueryPhone	Create an AC for a telephone device Flash the hookswitch on a telephone device Set the state of the hookswitch Returns the state of the hookswitch and loop current
I/O Control	AFEnableInput AFDisableInput AFEnableOutput ADisableOutput AFEnablePassThrough AFDisablePassThrough AFQueryInputGain AFQueryOutputGain AFSetInputGain AFSetOutputGain	Enable inputs on an audio device Disable inputs on an audio device Enable outputs on an audio device Disable outputs on an audio device Connect local audio to the telephone Remove the direct local audio/telephone connection Get minimum/maximum input gains for a device Get minimum/maximum output gains for a device Set the input gain of a device Set the output gain of a device
Access Control	AFAddHost AFAddHosts AFListHosts AFRemoveHost AFRemoveHosts AFSetAccessControl AFEnableAccessControl AFDisableAccessControl	Add a host to the access list Add a set of hosts to the access list Return the host access list Remove a host from the access list Remove a set of hosts from the access list Enable or disable access control checking Enable access control checking Disable access control checking
Properties	AFGetProperty AFListProperties AFChangeProperties AFDeleteProperty AFInternAtom AFGetAtomName	Manipulate properties Get a list of existing properties Modify a property Delete a property Install a new atom name Fetch the name of an atom
Error Handling	AFSetErrorHandler AFSetIOErrorHandler AFGetErrorText	Set the fatal error handler Set the system call error handler Translate error code to a string
Miscellaneous	AFNoOp AFFlush AFSync AFAudioConnName	Don't do anything Flush any queued requests to the server Default synchronization function Return the name of the audio server

Table 2: AudioFile client library functions

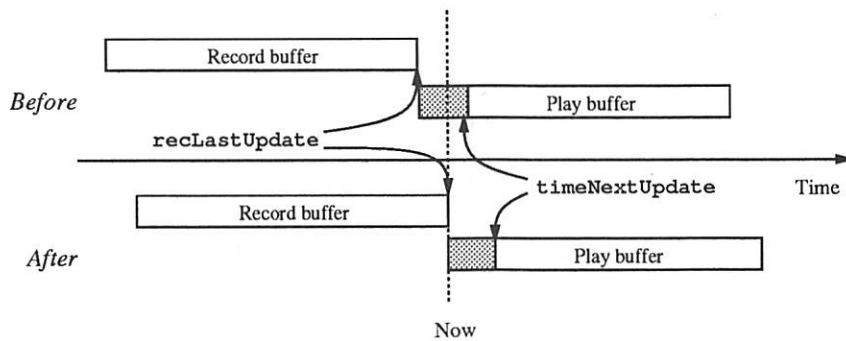


Figure 3: AudioFile periodic updates

7.3 Server Implementation

An AudioFile server is organized like an X server. It includes device independent audio (DIA), device dependent audio (DDA), and operating system (OS) components. The DIA section is responsible for managing client connections, dispatching on client requests, sending replies and events to clients, and executing the main processing loop. The DDA section is responsible for presenting the abstract interface for each supported device and contains all device-specific code. Finally, the OS section includes all the platform or operating system-specific code. Much of the OS and DIA code is based on X11R4.

Instead of using threads, we implemented a simple task mechanism which allows procedures to be scheduled for execution at future times, outside the main flow of control. The task mechanism is used by the server's update mechanism and by the dispatcher to resume execution of partially completed client requests.

At the core of the DIA section is the main control loop, which relies heavily on the `select()` system call. `select()` is called with file descriptors for client connections and open devices, as well as a timeout argument for the next task which needs to execute. When `select()` returns, the server runs any pending tasks and then handles input events and client requests.

Client requests are processed by the dispatcher. The request type is used to index into a table of protocol request handler procedures. All handlers are implemented by the device independent part of the server, but audio specific requests are passed to the device dependent audio server.

Adding DDA support for a new audio device is straightforward. The interface between the DIA and DDA sections of the server is very similar to that in the X server.

7.4 Device Dependent Server Examples

This section describes some of the device dependent implementation details of the LoFi, baseboard, and LineServer DDA code.

Alofi — LoFi Server

There are play and record circular buffers for each CODEC device and for each channel of the stereo HiFi device. For HiFi, we implemented a single stereo abstract device, as well as separate left and right devices for those clients not requiring stereo data. The host performs audio I/O by reading and writing these shared memory buffers. The LoFi's DSP runs a simple program that maintains the device time and buffers for each device.

We optimized the update procedure to achieve good performance. The server's periodic updates, which move samples between the server's buffers and the hardware, can consume quite a few CPU cycles, especially at high sample rates. We chose to have the HiFi record update run only if there is an active record client. The first record operation performed under an audio context marks the context as recording. A per-device reference count then controls the operation of the record update code. Similarly, the play update runs only when there is outstanding client data. We also initialize the server buffer with silence only when absolutely necessary. In the common case of contiguous playback requests, silence filling is never necessary.

Audio Handling	apass aplay arecord abiff abob radio xplay abrowse	Record from one AF server and playback on another Playback from files or pipes Record to files or pipes Incoming e-mail notification by audio Tk-based multimedia demonstration Multicast network audio X-based sound file browser Tk-based sound file browser
Device Control	ahs aphone aset aevents adial axset afxctl	Telephone hook switch control Telephone dialer Device control Report input events Tk telephone dialer Tk version of aset X-based event display and device control
Signal Processing Utilities	afft xpow autil	Tk based real-time spectrogram display X display of audio signal power Signal generator
Access control and Properties	ahost alsatoms aprop	AudioFile server access control Display defined atoms Display and modify properties

Table 3: AudioFile clients

Aaxp and Asparc — Baseboard Servers

The audio servers for the baseboard audio on Alpha AXP workstations and SPARCstations use kernel device drivers with similar interfaces. The server's update procedure uses the device driver read and write interfaces for recording and playing audio data. Because the kernel device drivers do not maintain a time register for the baseboard CODECs, the server maintains an estimated value using the system clock and must occasionally resynchronize with the device driver.

Als — LineServer Server

For the LineServer, an AudioFile server running on a nearby workstation uses a private UDP-based protocol to communicate with the device. The LineServer runs simple firmware that processes incoming packets and moves samples to and from the audio hardware. On the workstation, a periodic update task moves data between the server's buffers and the LineServer's buffers using the private protocol. The server makes every attempt to minimize access to the LineServer, since crossing the network is a relatively expensive operation. Only requests in the update regions require network traffic. For requests that require returning a device time, the server generates an estimate.

8 AudioFile Clients

The AudioFile distribution includes a number of client programs, including applications for recording, playback, and signal processing, as well as telephone, device, and access control. Some clients have graphical user interfaces using X and Tk toolkits [11]. Table 3 summarizes the clients. In the remainder of this section we describe two clients, aplay and apass, in some detail, to illustrate the simplicity of the AudioFile client library programming interface. We also include an answering machine shell script to show how simple AudioFile clients can be combined into larger applications.

8.1 aplay

aplay reads digital audio from a file or standard input and sends it to the server for playback. aplay can serve as the core of a sound-clip browser or voice mail retrieval program or as the final stage in a signal processing pipeline. For example, our software implementation of the DECTalk [3] speech synthesizer uses aplay for output.

aplay handles only "raw" sound files but could be easily extended to handle popular sound file formats. aplay works with any fixed-size encoding format and any number of channels — but the user must know the format of the

file and choose an appropriate server device.

As an example, we show a simplified version of `aplay`. Readers who wish all the details should read the sources, which are included in the `AudioFile` distribution.

```
aplay()
{
    aud = AFOpenAudioConn("");          /* open a connection to the server */
    device = FindDefaultDevice(aud);    /* select audio device */
    /* set up audio context, possibly setting the gain and endian-ness */
    ac = AFCreateAC(aud, device, (ACPlayGain | ACEndian), &attributes);
```

`FindDefaultDevice()` locates the lowest numbered audio device that is not connected to the telephone, which is usually the local audio device.

```
srate = ac->device->playSampleFreq;    /* sample rate */
type = ac->device->playBufType;         /* encoding type */
ssize = sample_sizes[type] * channels; /* bytes per sample */
buf = malloc(BUFSIZE*ssize);           /* allocate the play buffer */

nbytes = read(fd, buf, BUFSIZE*ssize); /* pre-read first buffer */
```

It is not logically necessary to pre-read the first file block, but doing so avoids putting the latency of the file read between the call to `AFGetTime()` and the first call to `AFPlaySamples()`.

```
t = AFGetTime(ac);                    /* obtain initial device time */
t = t + (time_offset * srate);        /* schedule initial playback */

do {                                  /* loop until done or brown on top */
    nact = AFPlaySamples(ac, t, nbytes, buf); /* send samples to the server */
    /*
     * At this point, the buffers in the server hold the samples from time
     * nact to time t. Next, we figure out how many samples we read
     * from the file, and schedule the next block to start after this one
     */
    t += (nbytes / ssize);
} while ((nbytes = read(fd, buf, BUFSIZE*ssize)) > 0);
```

This code fragment is the inner loop of `aplay`. It obtains the current device time and schedules the playback of the first block of audio. Thereafter, it schedules each successive block to play directly on the heels of the previous block, so playback will be continuous. After each call to `AFPlaySamples()`, the time pointer is simply incremented by the number of samples played.

There is no code in `aplay` for flow control. `aplay` makes a fundamental, but unwritten, assumption that the file system can supply audio data faster than it is required by the server. The audio data will be buffered in the server until `aplay` gets about four seconds ahead. At that point, the connection to the server will block occasionally, keeping `aplay` about four seconds ahead. The file I/O side of `aplay` can block for as long as four seconds without causing a break in the audio.

8.2 `apass`

Teleconferencing was one of our major goals in the development of `AudioFile`. `apass` is not a complete teleconferencing application; it simply records from one device and after a small delay, plays back on another device. However, `apass` addresses two of the fundamental problems of network teleconferencing:

- Management of end-to-end delay. In teleconferencing, it is important to have tight control over the end-to-end delay of the audio connection. If the round trip delay exceeds about 300 milliseconds, humans begin to have difficulty with conversational dynamics. `apass` sets up a strict delay budget, accounting for the various factors involved.
- Multiple clocks. In a system with multiple audio devices, the different devices are usually controlled by different clocks. If the transmitting end runs faster than the receiving end, then excess samples will accumulate in buffers in between, gradually increasing the end-to-end delay. If the transmit clock is slower than the receive clock, the buffers will run dry and the playback will break up. `apass` tracks the clock rates and resynchronizes as necessary.

apass reads blocks of samples from the transmit server and schedules their playback on the receive server. The delay between input and output is made up of three parts:

- Packetization Delay. The last sample of a block must be recorded before the first sample can be played back. The size of the block sets the minimum end-to-end delay.
- Transport Delay. apass reads samples from the transmit server and sends them to the receive server. The transmission delay, software overhead, and rescheduling delays make up the transport delay.
- Anti-Jitter Delay. apass uses explicit control over playback time to insert extra delay at the receiving server. This absorbs variation in the transport delay, provided that the variation is not larger than the anti-jitter delay.

```
apass()
{
    faud = AFOpenAudioConn(faf); /* open connections to the from */
    taud = AFOpenAudioConn(taf); /* and to audio servers */

    /* set up audio contexts, find sample size and sample rate */
    fac = AFCreateAC(faud, fdevice, ACRecordGain, &attributes);
    srates = fac->device->playSampleFreq;
    ssize = sample_sizes[fac->device->playBufType] * fac->device->playNchannels;
    tac = AFCreateAC(taud, tdevice, ACPlayGain, &attributes);

    /* establish a value for the delay from record to playback */
    delay_in_samples = fsrate * delay_in_seconds;

    ft = AFGetTime(fac); /* get starting times */

    /* playback will start delay_in_samples in the future */
    tt = AFGetTime(tac) + delay_in_samples;

    for (;;) {
        /* record samples and play them back */
        factt = AFRecordSamples(fac, ft, samples_bufsize*ssize, buf, ABlock);
        tactt = AFPlaySamples(tac, tt, samples_bufsize*ssize, buf);
```

AFRecordSamples() and AFPlaySamples() accept the parameters ft (from-time) and tt (to-time) respectively, and return factt (from-actual-time) and tactt (to-actual-time). factt, the current transmit server time, will be approximately equal to ft+samples_bufsize, because the pacing flow control is provided by the transmit server.

```
        est_delay = average_recent(tt - tactt); /* average recent delay estimates */

        /* if the delay has drifted outside of the allowable
           region, then resynchronize the connection */
        if ((est_delay < delay_lower_limit) || (slip >= delay_upper_limit))
            tt = tactt + delay_in_samples;

        /* finally, update the start times of the next block */
        ft += samples_bufsize; tt += samples_bufsize;
    }
}
```

apass uses a very simple algorithm for synchronizing clocks. $tt - tactt$ is an estimate of the current end-to-end delay minus the packetization and average transport delays. This difference should be about delay_in_samples, but will vary from this nominal value. apass averages several recent estimates to determine if the end-to-end delay is within the range specified by delay_in_samples plus or minus the anti-jitter specification. If the receive clock is too fast, the average delay will eventually drift below the lower end of the range. If the receive clock is too slow, then the delay will drift above the upper end of the range. In either case, tt is reset to nominal delay, resynchronizing the connection.

Note that time values from the two audio servers cannot be directly compared because they have different initial values and slightly different rates. apass avoids this problem by tracking the buffering available at the receiving server.

A robust teleconferencing application would likely choose a more sophisticated algorithm for managing multiple clocks, such as resynchronizing whenever the audio is quiet, or resampling at the receive sample rate.

8.3 A Trivial Answering Machine

Separate simple clients can be used to construct interesting applications. For example, we can implement a simple answering machine as a shell script.

```
#!/bin/sh
#
#   while true; do
#
#       aevents -ringcount 3.0           # wait for the phone to ring three times
#       ahs off                          # answer the phone (take off-hook)
#       aplay -f -d 0 outgoing_message.snd # play outgoing message
#       aplay -f -d 0 beep.snd           # play a beep
#
#   # record up to 30 seconds, or until the caller stops talking
#
#       arecord -silentlevel -35.0 -d 0 -silence 4.0 -l 30.0 -t -1.0 >>messages.snd
#       aplay -f -d 0 thanks.snd         # play a thank-you message
#       ahs on                           # hang up the phone
#
#   # add a date stamp to the message file using a text to speech
#   # synthesizer (not part of AudioFile ...)
#
#       date | tts >>messages.snd
#   done                                # Go back and get the next message
```

8.4 Other AudioFile Applications

There are already several applications which use AudioFile but are not distributed with it. We mention some interesting examples here.

DECtalk We built a software-only version of the DECtalk text-to-speech synthesizer, called *tts*. The synthesizer generates waveform samples on the standard output, which we can pipe to *aplay* for output.

DECspin DECspin is a Digital product for network audio and video teleconferencing. DECspin uses AudioFile to provide its voice capability.

VAT A team at the University of California, led by Van Jacobson, has built a network teleconferencing application using IP multicast protocols [4]. VAT can use AudioFile for its audio I/O.

Sphinx Sphinx II is a continuous speech recognition system [6] developed at Carnegie Mellon University. We use AudioFile's high fidelity input capability to supply audio to Sphinx.

9 Performance Results

In this section, we present some performance results for our implementation of the AudioFile System. First, we measure the time to complete client library operations. Next, we measure the CPU load for recording and playback. Finally, we briefly discuss our experience using TCP as the transport protocol.

9.1 Server and Client Performance

We measured latencies and performance of our AudioFile implementation by timing various client library functions. We tested with two types of systems (MIPS and Alpha AXP) under six local and networked configurations:

alpha	Alpha local client & server
alpha/mips	Alpha client, MIPS server
alpha/alpha	Alpha networked client & server
mips	MIPS local client & server
mips/mips	MIPS networked client & server
mips/alpha	MIPS client, Alpha server

We did all testing with the LoFi server, Alofi, using the 8 KHz CODEC device. All MIPS systems were DECstation 5000/200s running ULTRIX 4.3, and all Alpha AXP systems were DEC 3000/400s running DEC OSF/1 for Alpha AXP V1.2. All network testing took place on a lightly loaded 10 Mbit/sec Ethernet.

Play and Record

The AudioFile library functions that move data have latencies that depend on the length of the data. Figures 4 and 5 show the elapsed time for various length *AFRecordSamples()* and *AFPlaySamples()* requests on the different system

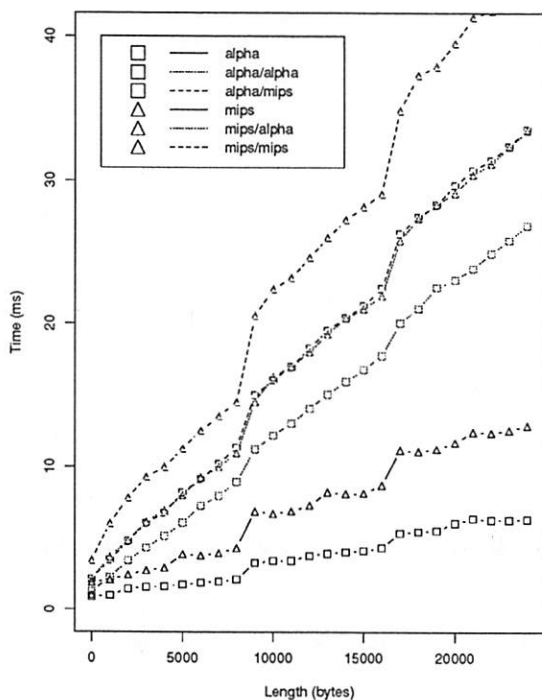


Figure 4: AFRecordSamples() timings

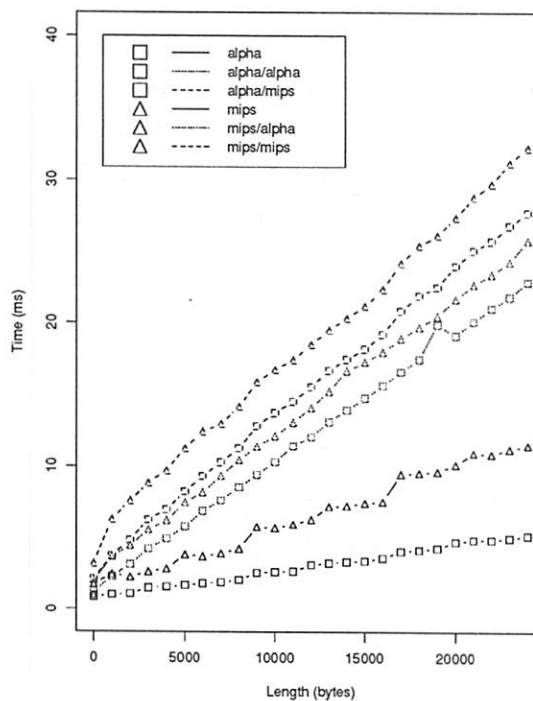


Figure 5: AFPlaySamples() timings

configurations. The requests were scheduled to hit entirely in the server's buffers (and therefore not block). These times were computed by computing the average time for 1000 operations.

The timings for short requests represent the basic overhead for a server/client exchange. In the record case, the jumps at approximately 8K bytes are due to "chunking" performed in the client library. Record requests larger than 8K bytes (not samples) are broken into 8K byte chunks to better control interactions with transport protocol heuristics and to simplify server implementation. Each record request completes synchronously — the client library waits for the reply before sending the next chunk. A 16K byte request therefore takes the same time as two independent 8K byte requests.

Play requests are chunked in a similar fashion, but the client does not wait for each chunk to complete. The client sends all chunks to the server and waits only for the reply to the last one. The resulting play timing is a nearly linear function of the play request size.

Open Loop Record/Play

The timings of various AudioFile operations have implications for applications that process audio in real time. Simple applications, such as playing a file, do not really care how long the operations take to complete, as long as the throughput exceeds the audio data rate. However, other applications, such as teleconferencing, depend on minimizing the time needed to handle samples.

To illustrate some of the implementation limits, we coded a loopback test that reads samples from a device and writes them back as quickly as possible. The test uses a non-blocking record function that returns only what samples are available. The algorithm is shown in this code fragment:

```
for(;;) {
    now = AFRecordSamples(ac, next, 8000, buffer, ANoBlock);
    length = now - next;
    AFPlaySamples(ac, next+4000, length, buf);
    next = now;
}
```


The rate at which this loop iterates is governed entirely by the AudioFile overhead and represents a limit for handling real-time audio. The average times to complete one iteration are shown in Table 4.

Configuration (client/server)	Time (ms)
alpha	0.87
alpha/alpha	1.27
alpha/mips	2.17
mips	1.93
mips/alpha	2.15
mips/mips	3.45

Table 4: Loopback timing

AudioFile's overhead establishes a minimum latency for real-time applications. However, we feel that AudioFile will be adequate for all but the most demanding real-time requirements. In some networked configurations, however, AudioFile's overhead will be small compared to the network delays. For example, a link across North America has a minimum 15 millisecond propagation time, not including transmission and routing time.

9.2 CPU Usage

We also investigated the CPU usage for playback and record operations. The tests consisted of playing and recording 30 seconds of audio at two sample rates and data types: 8 KHz μ -law and 44.1 KHz CD-quality stereo. Tests were done in a local configuration with the server and client running on the same machine, using UNIX domain sockets. Table 5 summarizes the server and client CPU usage for the two cases. Both user and system times (in seconds) are given, and the percentage column indicates the system load, computed by dividing total CPU time by the duration. These results include the costs of transferring data to LoFi using programmed I/O. CPU usage would be reduced by use of shared memory transport or by a DMA I/O device.

System	Test case	Server		Client		Total	% Load
		User	Sys	User	Sys		
Alpha DEC 3000/400	8 KHz playback	0.2	0.1	0.0	0.1	0.4	1.2
	8 KHz record	0.1	0.1	0.1	0.1	0.5	1.5
	44.1 KHz playback	2.9	2.0	0.0	0.5	5.4	18.0
	44.1 KHz record	3.7	1.1	0.7	1.4	6.9	22.9
MIPS DEC 5000/200	8 KHz playback	0.2	0.2	0.0	0.1	0.5	1.7
	8 KHz record	0.2	0.1	0.2	0.1	0.6	2.0
	44.1 KHz playback	1.9	1.9	0.0	2.6	6.4	21.3
	44.1 KHz record	4.4	4.0	1.1	3.4	12.9	43.0

Table 5: CPU usage for playback and record operations

9.3 Data Transport

AudioFile can be used over almost any transport protocol, though the details of the protocol may affect real-time audio performance. This section discusses our experience using TCP as the transport layer.

Although applications such as *apass* may exercise tight control over timing, most do not have strong real-time requirements. TCP is usually sufficient for these applications because the delay caused by retransmission of lost packets is small compared to the buffering of unplayed samples. On the other hand, applications like teleconferencing do require timely delivery of the audio data.

We found that a naively implemented teleconferencing application displayed serious problems when used over a transcontinental TCP link. We observed frequent and lengthy dropouts in the audio stream, which were especially likely with bidirectional data streams. These stem from packet losses caused by a phenomenon known as "ACK-compression" [10, 20], a subtle consequence of the use of window-based flow control. The duration of each dropout

is exacerbated by TCP's slow-start algorithm [5], which comes into play when packets are dropped by the network.

ACK-compression occurs when the spacing between acknowledgments is changed by delays in the routers. This can cause TCP to send large bursts of packets, which overrun the buffers in a router, causing packets to be dropped. Unfortunately, the TCP slow-start algorithm converts these losses into lengthy recovery periods during which data flows more slowly. On a connection such as a long-haul T1 circuit, it can take several seconds to restore full throughput.

TCP is arguably the wrong transport protocol for applications such as teleconferencing, since it tries to guarantee ordered packet delivery without any concern for packet delay. Many applications instead need guarantees on bandwidth and latency, but they may be prepared to accept some lost data. Networks and protocols that provide such guarantees are active areas of research. To manage these issues, all of the teleconferencing applications mentioned in Section 8 are split among sites, using special protocols over long-haul paths, and only communicate locally with AudioFile servers.

10 Summary

The AudioFile System provides device-independent, network-transparent audio services. With AudioFile, multiple audio applications can run simultaneously, sharing access to the actual audio hardware. Network transparency means that application programs can run on machines scattered throughout the network. Because AudioFile permits applications to be device-independent, applications need not be rewritten to work with new audio hardware.

10.1 Areas for Further Work

It is remarkably difficult to get something as big as AudioFile completely right. We are very pleased with our basic design decisions, but we do have a list of items which, if implemented or fixed, would make AudioFile still more useful.

- Audio devices should support multiple sample rates, and the library should support sample rate changing.
- Audio devices should have an ordered list of supported data formats, so that the device can express a preference for one format over another.
- The protocol and library should offer improved support for synchronization and conversion between clocks, including clock prediction routines and the simultaneous reporting of all device clocks.
- Clients should be able to refer symbolically to "the local loudspeaker" or "the telephone".
- Clients which deal with files should know about various popular sound file formats.
- The server should support compressed data types, which would make AudioFile more useful in low-bandwidth environments.

10.2 Conclusions

We believe that AudioFile has done well in meeting our design objectives: network transparency, device independence, simultaneous clients, simplicity, and ease of implementation. We also believe our experience has validated our principles:

- Client control of time. AudioFile permits both real-time and non-real-time audio applications using the same primitives.
- No rocket science. Our decisions to build on top of standard communications protocols and not to use threads have improved the portability of the system. It is also arguable that AudioFile performs so well precisely because of its minimalist underpinnings.
- Simplicity. Simple play and record clients require very little code. Indeed, many applications can be constructed using independent AudioFile clients organized by shell scripts.
- Computers are fast. We did not let fear of per-sample processing get in our way. Our slowest implementation uses less than 2% of the CPU for telephone-quality playback.

10.3 How to Get AudioFile

AudioFile can be copied by anonymous FTP from `crl.dec.com` (192.58.206.2) in `/pub/DEC/AF/AF2R2.tar.Z`. Some stereo sound bites are in `AF2R2-other.tar.Z`. A more detailed description of the design and implementation of AudioFile is also available as a technical report [8].

We apologize for the code being cluttered with left-justified chicken scratches.⁸ We wanted AudioFile to be portable, but at least one major vendor does not support function prototypes with their stock C compiler.

We have created an Internet mailing list `af@crl.dec.com` for discussions of AudioFile. Send a message to `af-request@crl.dec.com` to be added to this list.

10.4 Acknowledgments

Many people have contributed to AudioFile. We would like to thank Ricky Palmer and Larry Palmer for the SPARCstation DDA code. They, along with Lance Berc and Dave Wecker, persevered as early users of AudioFile. Lance Berc's experiments with long-distance teleconferencing taught us much about the network issues. Jeff Mogul offered valuable assistance on understanding these issues. Guido van Rossum contributed the DDA code for the Silicon Graphics Indigo only two weeks after we released the first public distribution. Dick Beane provided useful comments on drafts of this paper. We would also like to thank Victor Vyssotsky and Mark R. Brown for putting up with us.

References

- [1] Susan Angebrannt, Richard L. Hyde, Daphne Huetu Luong, Nagendra Siravara, and Chris Schmandt. Integrating audio and telephony in a distributed workstation environment. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.
- [2] B. Arons, C. Binding, K. Lantz, and C. Schmandt. The VOX audio server. In *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, 1989.
- [3] Edward Bruckert, Martin Minow, and Walter Tetschner. Three-tiered software and VLSI aid developmental system to read text aloud. *Electronics*, Apr. 21, 1983.
- [4] Steve Deering and Steve Casner. The first IETF audiocast. *ACM Communications Review*, 22(3), July 1992.
- [5] Van Jacobson. Congestion avoidance and control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, August 1988.
- [6] Kai-Fu Lee. *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Academic Publishers, Norwell, MA, 1989.
- [7] Thomas M. Levergood. LoFi: A TURBOchannel audio module. CRL Technical Report 93/9, Digital Equipment Corporation, Cambridge Research Lab, 1993.
- [8] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart. Audiofile: A network-transparent system for distributed audio applications. CRL Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab, 1993.
- [9] D. L. Mills. Network time protocol (NTP). Internet RFC 958, Network Information Center, September 1985.
- [10] Jeffrey C. Mogul. Observing TCP dynamics in real networks. In *Proc. SIGCOMM '92 Symposium on Communications Architectures and Protocols*, Baltimore, MD, August 1992.
- [11] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the USENIX Winter Conference*, January 1991.
- [12] Steven J. Rohall. Sonix: A network-transparent sound server. In *Proceedings of the Xhibition 92 Conference*, June 1992.
- [13] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, Bedford, MA, 3rd edition, 1991.

⁸You really should read Thompson's paper on the Plan 9 C compiler [18].

- [14] Henry Spencer. How to steal code -or- inventing the wheel only once. In *Proceedings of the USENIX Winter Conference*, pages 335-346. USENIX, February 1988.
- [15] D. C. Swinehart, L. C. Stewart, and S. M. Ornstein. Adding voice to an office computer network. In *Proceedings of GlobeCom 1983*, November 1983.
- [16] Robert Terek and Joseph Pasquale. Experiences with audio conferencing using the X window system, UNIX, and TCP/IP. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.
- [17] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909-920, Aug. 1988.
- [18] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conf.*, pages 41-51, London, July 1990.
- [19] Stephen A. Uhler. PhoneStation, moving the telephone onto the virtual desktop. In *Proceedings of the USENIX Winter Conference*. USENIX, January 1993.
- [20] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proc. SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pages 133-147, Zurich, September 1991.

Author Information

Lawrence C. Stewart joined the Cambridge Research Lab (CRL) in 1989 after 5 years at Digital's Systems Research Center. His interests include speech, audio, and multiprocessors. He was one of the designers of the first Alpha AXP computer system. Before joining Digital, Larry was at Xerox PARC. He received an S.B. from MIT in 1976, and M.S. and Ph.D. degrees from Stanford in 1977 and 1981, all in Electrical Engineering.

G. Winfield Treese joined CRL in 1988 after working at MIT on Project Athena. Win's interests are in networks and distributed systems. He received an S.B. in Mathematics from MIT in 1986 and an S.M. in Computer Science from Harvard University in 1992. He is now pursuing a Ph.D. at MIT in the area of computer networks.

James Gettys joined CRL in 1989. Jim's focus at CRL is multimedia audio and video systems. Before joining CRL, Jim spent two years at the Systems Research Center. Before that, he was a Digital engineer and visiting scientist at MIT working on Project Athena. He is one of the two principal designers and developers of the X Window System. Jim received an S.B. from MIT in 1978.

Andrew C. Payne joined CRL in 1992 after receiving a B.S. in Electrical Engineering from Cornell University. As a co-op at Digital in 1990, he helped build the first Alpha AXP chip. Andy's interests include signal processing, speech, and user interfaces.

Thomas M. Levergood joined CRL in 1990. Tom's focus is on speech and audio-related research. He is also involved in Alpha AXP system and software projects, including an experimental evaluation of split user/supervisor cache memories. He received his B.S. in Electrical Engineering in 1984 and M.S. in Electrical Engineering in 1993, both from Worcester Polytechnic Institute.

All of the authors can be reached at: Digital Equipment Corporation, Cambridge Research Lab, One Kendall Square, Bldg. 650, Cambridge, MA 02139, or by e-mail as {stewart, treese, jg, payne, tml}@crl.dec.com.

Fast and Flexible Shared Libraries

Douglas B. Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg

University of Utah

Abstract

Existing implementations of shared libraries sacrifice speed (in loading, linking, and executed code), for essential flexibility (in symbol binding, address space use, and interface evolution). Modern operating systems provide the primitives needed to make the dynamic linker and loader a persistent *server* which lives across program invocations. This can provide speed without sacrificing flexibility. The speed is gained primarily through caching of previous work, i.e., bound and relocated executable images and libraries. The flexibility comes from the server's being an active entity, capable of adapting to changing conditions, modifying its cached state, and responding to user directives. In this paper we present a shared library implementation based on OMOS, an Object/Meta-Object Server, which provides program linking and loading facilities as a special case of generic object instantiation. We discuss the architecture of OMOS and its support of module binding primitives, which make it more flexible and powerful than existing shared library schemes. Since our design does not require any support from the compiler, it is also language-independent and highly portable. Initial performance results, on two operating systems, show an average speedup of 20% (range 0 - 56%), on short running programs.¹

1 Introduction

The typical implementations of shared libraries hold different places on the static—dynamic spectrum. On one extreme we find the original System V shared libraries which need all resources to be fixed at link time — including the location of the libraries. On the other extreme we find shared library implementations whose symbols are resolved dynamically at run time.

Clearly, the former performs well because there is little work that must be done at run time — but there is little flexibility in this approach. Dynamic solutions are usually slower because there is more work to be done. But it is often the case that the work done is a repetition of work already performed — if the same memory image is produced 20 times, the work done for the last 19 is unnecessarily repeated in order to retain flexibility.

If we approach the object and executable files as the only possible sources of executable instructions on a system, our choices are limited. These files are static entities with limited semantics, constraining their flexibility. In principle, the only version of a program that matters is the image executing in memory (the final implementation). If we view the object file merely as a convenient intermediate form, our options become more interesting.

We describe a scheme which provides the flexibility of dynamic approaches while retaining the speed of static designs, and in some cases, improving on it. This design is based on OMOS, a server which is capable of dynamically generating executable images. As a server, OMOS is capable of reacting to changing conditions and maintaining the flexibility of traditional shared libraries, without requiring special compiler

¹This research was sponsored in part by the Hewlett-Packard Research Grants Program and by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of ARPA, the U.S. Government, or Hewlett-Packard.

support. OMOS treats executable images as a cache, translating from more expressive forms (e.g., .o's, or source modules) as necessary. By treating executables as a cache, OMOS avoids unnecessary repetition of work. As a natural side effect of its cache management, OMOS provides the sharing found in shared library systems.

2 Shared Libraries

2.1 Benefits and Caveats

The traditional benefits of shared libraries are well known. Their original primary motivation derived from their *shared* aspect, which reduced main memory use. This was the driving motivation some years ago, when memory was more of a scarce resource. Sharing also reduces disk consumption and eases software maintenance, since a library fix is instantly incorporated into all clients of that library. As shared library implementations became more sophisticated with support for *dynamic binding*, this allowed their exploitation for environment configuration. For example, internationalization is often supported through an environment variable.

However, shared libraries have other important benefits which are not as widely recognized. An apparently mundane one, but very important in a development environment, is drastically reduced static linking time. Statically linking a multi-megabyte binary, not at all unusual with the huge libraries now existing, can take many minutes. This becomes at least a factor of three worse when writing to a traditional NFS implementation, in which writes are synchronous. The lazy procedure binding used by most current shared library schemes, makes acceptable the run time performance of shared library versions of such programs. However, the majority of the savings probably comes from avoiding the I/O in writing out huge binaries.

Another less frequently recognized and exploited benefit, but of great importance and power, is the flexible binding afforded by modern shared library implementations. When procedure binding is dynamic, one can, in principle, *substitute* a different procedure at run time, or interpose a *wrapper* procedure around the original. This provides powerful flexibility, separating interface from implementation. This capability has been noted before[7], as well as the similarity between objects and shared libraries[18].

However, existing implementation of shared libraries have significant problems. They offer only limited control over the flexible binding discussed above. The working set sizes of programs increase, because the library functions an application uses are scattered across a large set of pages with other, unused functions. Memory savings can be minimal, at least with small libraries, due to the size of the dispatch tables[11]. In general, the memory savings from shared libraries are probably more significant in a multi-user time-shared system than in the dedicated workstation environment common today, though there are no good measurements of this.

2.2 Issues

The traditional issues in shared library design and implementation are well recognized. For a complete discussion, refer to the articles by Sabatella[15] and Gingell[8], who give good overviews. These include granularity of sharing, granularity of symbol binding, code purity and the methods to achieve it, address binding, time of binding, time of loading, and version control. Additional issues have since surfaced, including handling languages such as C++ which present special problems such as static initializers[16].

3 Overview of OMOS

3.1 Overview

The OMOS object/meta-object server is a process which manages a database of *objects* and *meta-objects*. Objects are code fragments, data fragments, or complete programs. These programs may embody familiar services such as *ls* or *emacs*, or they may be functions or objects providing library-like services such as a

binary tree object. Meta-objects are templates describing the construction and characteristics of objects, and contain a class description of their target objects.

OMOS uses the meta-object to construct and load implementations of a given class into client address spaces. For example, given a meta-object for `ls`, OMOS can create an instance of the `ls` class for a client. Instantiating an object subsumes linking and loading a program in a more traditional environment. By caching intermediate results of the construction process, OMOS can avoid unnecessarily reproducing effort when instantiating program binaries (e.g., parsing executable file formats, setting up mappers, etc.). When several applications within OMOS share sub-objects (such as those that make up conventional libraries), the effect is to share the underlying physical memory. In this way, by providing a shared object service, OMOS indirectly provides a shared library service.

3.2 OMOS and Class Instantiation

Client programs communicate with OMOS via an IPC mechanism, requesting instantiation of classes via a meta-object specification. OMOS maintains and exports a hierarchical namespace, whose names represent meta-objects, executable code fragments, or directories of other objects. Typically, clients request instantiation of meta-objects by name. An instantiation request may be accompanied by a *specialization*, indicating specific qualities the resultant implementation is to possess (including adherence to a given address space layout, required program facilities, etc.). If OMOS has a cached image associated with that meta-object that meets the user's specifications, the image is returned. If no cached image exists, OMOS constructs one by following the plan encoded within the meta-object. Ultimately, the cached image is mapped into the user's address space.

Meta-objects contain a specification, known as a *blueprint*, which describes how to combine objects and other meta-objects to produce an instance of the class. These rules map into a graph of operations, the *m-graph*. To generate an executable image, OMOS parses the blueprint and constructs an m-graph. The m-graph is executable; execution of the m-graph will generate an implementation of the class. Before executing the m-graph, OMOS applies any user-specified specializations to it, transforming the m-graph as appropriate.

Execution of the m-graph may result in OMOS compiling source code, performing symbol translations, and combining and relocating fragments. An m-graph operation may have as its operand other m-graph operations; the operation may in turn expand and execute its arguments. After successful execution of the m-graph, the resultant mappable image is cached and returned to be mapped into the user's address space.

3.3 OMOS Blueprints and the Jigsaw Operators

Since one of OMOS' primary functions is to link and load programs, an important focus of the blueprint operations is manipulation of symbol namespaces. A subset of the graph operations comprise *module operations*, as defined by Bracha and Lindstrom in the language *Jigsaw*[3, 4]. Jigsaw was designed in order to *decompose* the bundled module definition and manipulation found in programming languages, including the manipulation performed in *inheritance*.

Conceptually, a module is a self-referential naming scope. Module operations operate on and modify the symbol bindings in modules. The modified bindings define the inheritance relationships between the component objects. The Jigsaw operators can be naturally composed to perform function interposition and inheritance as well as overriding or renaming of functions or data. The set of graph operations into which a blueprint may be translated is described in more detail in Section 3.3.

An advantage of representing the object construction as an executable graph is that it lends itself naturally to program transformations. A given operation may transform all or part of any of its operands. The specialization phase of evaluation may transform an entire meta-object. The sorts of transformations OMOS is capable of performing range from transparent interposition of monitoring routines to constraining objects to be bound within certain address ranges. Since OMOS is an active entity, the types of transformations it performs may change based on feedback or changes in the execution environment (as in the case of program monitoring where the execution of the program changes the implementation OMOS generates).

Currently, the specification language used by OMOS has a simple Lisp-like syntax. The first word in an

expression is a graph operation (described below) followed by a series of arguments. Arguments can be the names of server objects, strings, or other graph operations. Each operation produces a module as its output. In this example

```
(merge
  /lib/crt0.o /obj/ls.o
  /lib/libc)
```

the `ls` program is constructed by merging (linking) two fragments with another meta-object, which represents the Unix library `libc`.

As indicated, m-graphs are composed of nodes which may contain graph meta-objects, or fragments. The complete set of graph operators defined in OMOS is described in [13]. The graph operators important to this discussion include:

Merge: binds the symbol definitions found in one operand to the references found in another. Multiple definitions of a symbol constitutes an error.

Override: merges two operands, resolving conflicting bindings (multiple definitions) in favor of the second operand.

Freeze: makes permanent a given symbol binding.

Restrict: virtualizes a set of bindings: any existing bindings become unbound and any existing definition for the symbol are removed.

Project: is the complement of restrict. project virtualizes all but a given set of bindings.

Copy-as: duplicates the value of a symbol definition under a new name.

Hide: removes a given set of symbol definitions from the operand symbol table, freezing any internal references to the symbol in the process.

Show: is the complement of hide which hides all but a given set of symbol definitions.

Rename: systematically changes names in the operand symbol table. Names may be references, definitions, or both.

Initializers: generates C++ static initializers for the C++ objects found in the file.

Specialize: specializes the operand to behave in a particular fashion (e.g., as either a fixed-address or dynamically loaded shared library, etc.)

Source: produces a fragment from a C, C++, or assembly language source object.

Constrain: directs OMOS to try to use or avoid a given address region when loading an object. The result of this operation can be mapped into a user address space.

List: groups two or more server objects into a list.

The leaf operands to OMOS operations are relocatable object files. OMOS manipulates relocatable object files using an idealized interface for symbol manipulation. High-level OMOS operations eventually translate into operations on an object file symbol space. OMOS provides a facility that allows many different name configurations ("views") to be mapped onto a given object file, allowing fast, efficient, incremental modification of a symbol namespace. Module operations typically take a regular expression as a specification of the symbols to select. Execution of a module operation (with the exceptions of `merge` and `freeze`) results in the production of a new view of the operand.

3.4 OMOS and Specialization

A powerful aspect of the OMOS framework is the use of executable graphs to represent the operations to be performed when constructing class implementations. Specialization adds another level of flexibility to the use of meta-objects. A base meta-object, representing an idealized class specification, can be transformed

in various ways by OMOS. Using specialization, OMOS translates the underlying m-graph into a new, specialized m-graph.

For example, OMOS uses specialization to choose the style of shared library to generate from a given meta-object. As described in Section 4, OMOS supports two basic styles of shared libraries, with very different invocation mechanisms. The base implementation and the abstraction of the library is the same in both cases. To clients of the different schemes, only the invocation mechanism differs.

OMOS manages these differences as cases of specialization of a base library implementation. The user links through a special meta-object that transforms the target library into the appropriate form. The concept of the idealized library is maintained. For example, the operation:

```
(specialize "lib-dynamic" /lib/libc)
```

produces a library with dynamic routine resolution, while the operation:

```
(specialize "lib-constrained" (list "T" 0x1000000) /lib/libc)
```

produces a fixed-address version of libc whose text segment is constrained to live as close to the address 0x1000000 as possible. Merging with the first operation would result in the client's being linked with a set of stubs that execute at run time. Merging with the second operand would result in a client that is bound directly to an implementation of libc chosen by OMOS' constraint system. Both instances represent the notion of merging a client with an abstract "libc," although the specific implementations are very different.

3.5 OMOS and Constraints

OMOS uses its specialization facility to maintain flexibility in its shared library implementation. The address constraint specialization is a simple, yet powerful mechanism for guiding the placement of libraries within an address space.

OMOS describes an address space in terms of prioritized constraints. A required constraint is that no two objects may overlap. A highly desired constraint is that existing implementations be reused. Other weaker constraints, optionally provided by the user, may specify desired placement of the object (e.g., library) within the address space. When no existing implementation meets all the given constraints, OMOS will generate (and cache) a new one. If an existing implementation meets the given constraints, OMOS will reuse it, causing its read-only portions to be shared among its different clients.

In the case that a user attempts to make use of an address space region that is in conflict with existing libraries, the constraint system considers the priorities associated with each requirement and resolves the problem by using alternate implementations or generating new ones. Subsequent invocations of the same combination of applications and libraries will use the existing set of implementations.

4 OMOS Shared Library Schemes

As previously mentioned, OMOS supports different strategies for implementing shared libraries. A *library* class, derived from the standard meta-object class, is used to describe OMOS shared libraries. The class specifies a default specialization to be applied to the meta-object which may be used to generate different implementations. In figure 1 we see a sample implementation of the library "libc". The first line specifies the default specialization to be applied to the implementation (producing a self-contained shared library in this example). The rest of the meta-object describes how to construct the base implementation of the library.

4.1 Self-contained Shared Libraries

In the standard scheme for implementing shared libraries in OMOS, OMOS maintains both the client application and library as objects internal to itself. OMOS makes use of its constraint system to allow


```
(constraint-list "T" 0x100000 "D" 0x40200000) ; default address constraint
(merge
  /libc/gen /libc/stdio /libc/string /libc/stdlib
  /libc/hppa /libc/net /libc/quad /libc/rpc)
```

Figure 1: Sample Library Meta-Object

different address space configurations to be used, without explicit intervention of the user or a system manager.

In this scheme, each implementation of an application that references a library will be bound to a particular version of the library. The library version will be located at a fixed address. As a result, all resolution of addresses and linking is done at the time a fixed version of the application is constructed. Typically, in a development environment, fixed versions of clients and libraries are generated at installation time, to avoid the startup latency on first invocation.

If the application meta-object specifies a library for which there is no implementation, or for which the existing implementations conflict with other address space requirements, OMOS will use its constraint system to resolve the conflict. OMOS' constraint system will cause new versions of some number of libraries to be generated in a configuration in which there are no conflicts. In the common case only one implementation of each library will ever be generated. Currently, a little planning by the system manager helps optimize this. However, OMOS could easily record the conflicts found, and occasionally the system manager could feed that data into OMOS' constraint system to determine better placements, or this could be done fully automatically. (It should be noted that it is fairly important that few versions of libraries be generated, since disk space for caching multiple versions of large libraries could be significant.)

In this way, OMOS achieves the same flexibility as dynamic shared library schemes based on position independent code (PIC), as well as the traditional sharing benefits, without the run time overheads of dynamic binding. By using its active nature, OMOS can specialize its implementations to the actual address requirements of its clients, rather than relying on the availability of a completely general framework such as PIC.

As is normal with shared libraries not using PIC, variables shared between application and libraries, or between two libraries, can pose problems in this scheme. If shared variables are defined in the client application, and referenced by the library, the physical sharing of the library by two different applications can not be maintained. References may exist in the client application, but, in general, all definitions of variables must be made in the library "furthest downstream" that references them. No circular references may exist. Similarly, if libraries directly reference user procedures, OMOS will have to construct a new library image for each different application. OMOS or the system manager can monitor such occurrences, and if they are frequent, specialize the offending libraries to dispatch via a branch table.

The use of self-contained shared libraries poses inconveniences for debugging, since the program initially executed by the user may be a bootstrap loader whose only function is to load the client application and libraries. As a result, by default, the application debugging symbols will not be present in the file. With a little extra trouble (e.g., use of the `gdb` "symbol-file" command), the client symbols can be loaded in after the client and libraries have been mapped in. As a more elegant solution, we plan to enhance `gdb` to interface directly with OMOS, allowing them to interact seamlessly.

Self-contained Shared Library Speed and Memory Use

The self-contained shared library scheme benefits from simplicity. Linking work done to resolve references to library routines need not be repeated in order to preserve flexibility. In the common case, libraries can reside at a single location. Thus, this strategy tends to optimize the common case. The amount of work required to load a cached executable is constant, where schemes that do dynamic link resolution to maintain flexibility often must do work in proportion to the number of external references made by the client, every

time the library is loaded.

Since the self-contained shared libraries have no dispatch table, the absolute memory requirement for applications is decreased. For small programs, shared library dispatch tables can account for a surprising portion of program size. (Initial measurements of the SunOS implementation have shown that for small programs (e.g. `ls`) and libraries (`libc`), more memory is used for dispatch tables than is saved in library code[11].)

In addition, the self-contained shared library scheme can use absolute addressing modes to reference data, which has particular performance benefits, important on CISC machines but also relevant to RISC architectures. Use of the OMOS constraint system does not preclude the use of position-independent code, of course. In fact, the availability of PIC makes resolution of address constraints trivial, since there is effectively no cost to relocate an implementation. PIC is not required, however, to use the base functionality, which renders this scheme simple and portable.

OMOS also benefits from special optimizations that can be performed due to its being an active entity. One such optimization is reordering code based on function usage in order to improve locality of reference. OMOS can automatically generate implementations that will produce monitoring data, which it will then use to derive a preferred routine order. This reordering benefits both cache performance and paging behavior. We have performed this experiment and achieved average speedups in excess of 10%, as described in [14].

4.2 Partial-image Shared Libraries

An alternative scheme for shared libraries is the *partial-image* shared library, which provides more convenient application debugging facilities. In partial-image shared libraries, the client program consists of a complete copy of the application code, which is exported to the user (normally, OMOS manages all executable images as a cache and does not expose individual executables to the user).

The partial-image application contains stub routines for each library entry point. On the first invocation of a routine in a library, the client stub contacts OMOS and loads in the library, returning the address of a hash table containing the addresses of all library routines. The first time a function in a dynamically loaded library is accessed, its name is looked up in the function hash table and the value of its entry point is stored in an indirect branch table. Subsequent invocations of the function are made through the pointer in that table.

The *specialize* module operation provides a framework for implementing dynamically linked shared libraries within OMOS, using two types of specialization. The “lib-dynamic” specialization generates an m-graph which implements the application’s access to the library (i.e., the stubs which perform dynamic loading), and the “lib-dynamic-impl” specialization generates the m-graph which will produce the library implementation that is to be loaded and shared.

The “lib-dynamic” specialization creates an m-graph, the evaluation of which causes stub functions to be dynamically generated for each referenced entry point in the operand. The stub code is compiled and returned as the representative implementation of the library. On future invocations of the specialization the cached copy of the compiled function stubs will be used. The function stubs also contain a reference to the “lib-dynamic-impl” specialization of the library. On execution of the client application, the “lib-dynamic-impl” implementation of the library is loaded from OMOS.

Partial-image shared libraries allow developers to make use of unmodified system debuggers to debug OMOS client applications. They also eliminate the need by the user for knowledge of the mapping process. In addition, partial-image shared libraries provide a more traditional interface to OMOS. Since a dynamically linked application is an executable file, the semantic meaning of file rename, copy, and delete are the same as with statically linked programs. In debugging environments, developers may prefer the higher degree of control provided.

The utility of partial-image shared libraries is limited by the use of shared variables. Since the application is fixed and outside of OMOS’ control, references by the application to variables shared with libraries are problematic. Either the variables must live in a fixed segment whose address and size cannot change, or the application must use some level of indirection to access the shared variables. We could make use of

shared variables within this context less inconvenient with compiler modifications, but one of our portability goals was to avoid reliance on particular compiler features. As a result, the practical utility of partial-image shared libraries will tend to be limited to debugging or to “well-behaved” libraries whose shared state is encapsulated and accessed via procedures. *Versioning* should be implemented and would provide safety, but it would not help with resource consumption, i.e., disk space.

5 Constructing and Executing Programs Using OMOS

All programs constructed in OMOS have a meta-object specification. Users of a shared library reference the library name symbolically as part of a “merge” or “override” operation within the client meta-object. For example, the meta-object seen earlier,

```
(merge
  /lib/crt0.o /obj/ls.o
  /lib/libc)
```

will resolve references from `/lib/crt0.o` and `/obj/ls.o` to symbols found in the default implementation of library `/lib/libc`. Execution of this set of operations will ultimately produce mappable, executable images representing the main program and the library `libc`. If we give this meta-object a name, the mappable result will be accessible to clients of OMOS.

There exist several different mechanisms for instantiating classes maintained by OMOS. One client program of OMOS is a small bootstrap loader used to load OMOS meta-objects. In Unix, we normally invoke this loader via the “interpreter” feature (`#!/bin/omos`). This allows us to export entries from the OMOS namespace into the Unix namespace, in a portable fashion (as a parameter in the file). When invoked, the bootstrap loader contacts OMOS via IPC, loads in the executable image(s) for a given meta-object, and jumps to its entry point, subsuming the functionality of the system call `exec()`². In our example, if `/lib/libc` is a self-contained shared library, the bootstrap loader will map the main program in at one set of addresses, and the `libc` library at a different set. In the absence of address conflicts, all other clients of `/lib/libc` will share the same mapping and hence, the same physical memory as this client of `/lib/libc`. Another example of this mechanism is the partial-image scheme, in which applications contain the main program as well as code to contact the bootstrap loader.

However, use of the bootstrap loader involves a certain amount of overhead because the system must first load and execute the bootstrap before OMOS can do its work. Ideally, OMOS should be integrated into the operating system’s implementation of the `exec()` system call, which we have done in the OSF/1 operating system running atop Mach 3.0. When an object in OMOS’ exported namespace is requested to be executed, `exec` sets up an empty task and calls OMOS with handles to the task and the OMOS object. OMOS does its normal work, resulting in a set of mappable segments, which it then maps in to the target task. This replaces the portion of `exec` which is responsible for reading in object file contents. In general, OMOS should be able to do this more efficiently than the operating system, because it does not have to open files, parse complex object file headers, etc. Note that when OMOS is integrated in this manner, `/bin`, for example, can become a “filesystem” backed only by OMOS. The meta-objects and executable fragments providing the contents can be stored anywhere.

Program loading is a special case of the more general class generation and loading facility which OMOS provides. In addition to `exec()`-style invocation facilities, OMOS exports a more general interface for dynamically loading class implementations into executing programs. Via a meta-object, a client program specifies the class to be loaded, any specializations to apply to the meta-object, and a list of symbols whose bound values are to be returned from OMOS. The meta-object specification may either be the name of a meta-object found within the OMOS namespace, or an arbitrary blueprint to be executed by OMOS. This invocation mechanism is being used to load class methods in a project building a portable, persistent object management system on Unix[12].

²This is the invocation method we used to gather timings on HP-UX.

```

;;
;; malloc() -> malloc'()
;;
(hide "_REAL_malloc"
  (merge
    ;; Get rid of the old definition
    (restrict "~_malloc$"
      ;; stash a copy of _malloc() for later use
      (copy_as "~_malloc$" "_REAL_malloc"
        (merge /bin/ls.o /lib/libc.o)
      )
    )
    ;; Merge in a new definition
    /lib/test_malloc.o
  )
)

```

Figure 2: Interposition Example

As a note, execution of arbitrary blueprints can be used to implement a dynamic loading facility similar to that found in the dynamic linker, `dld[9]`. A client can request that new classes be loaded, which are then merged with its own implementation, allowing the new classes to refer to procedures and data structures within the client. A limitation of this scheme is that the client and target classes must both be within OMOS (not a problem in our view!) and the client must keep track of which classes it has dynamically loaded. We plan to add a facility to OMOS that will allow it automatically to maintain this information for interested clients.

Security is an obvious concern in the cases in which an unprivileged client invokes OMOS. In a system with a powerful IPC such as Mach's, this is not a problem. In normal Unix systems, the privileged port feature provides authentication that many utilities rely on—for OMOS to rely on it introduces no additional vulnerabilities. Thus the bootstrap loader can be `setuid`, providing an authenticated channel to OMOS. In general, the Unix file permission scheme can be used, applied to OMOS-managed objects.

6 OMOS Flexibility

The use of module operations and executable graphs by OMOS provides significant flexibility not found within conventional loading and linking environments. By using module operations grounded in theory, we provide complete symbol manipulation and binding facilities. The fact that OMOS is a server allows it to respond dynamically to program needs. That it is a server also makes it convenient to maintain program state across invocations.

Module operations can easily be used for interposing new routines within an executable. By invoking `copy-as` on all definitions of a given set of symbols using some well-known scheme (e.g., prepending a package name), then using `restrict` to virtualize the original bindings, new values for the symbols in question can be inserted transparently in the original application.

For example, in Figure 2, we produce a version of the C library, `libc`, where a new version of `malloc` has been inserted to trap calls to the original routine. References to the native routine in the new routine are preserved.

The `source` operator can be used to fill in missing variable or routine definitions with default values. The `rename` operation can be used, as seen in Figure 3, to do something more drastic, such as to rename all references to routines that should never be called to the routine `_abort`, which will produce notable behavior if called unintentionally.

```

(merge
  ;; resolve an undefined data reference and
  ;; reroute undefined routines to "abort()"
  (source "c" "int undef_var = 0;\n")
  (rename "^_undefined_routine$" "_abort"
    /lib/lib-with-problems))

```

Figure 3: Symbol Renaming and Resolution Example

As mentioned in Section 4.1, OMOS can transparently modify program executables to provide monitoring data, which can later be used to reorder the application to improve performance. OMOS does this by using module operations to extract the set of referenced routines and generate wrapper functions around each, to log entry and exit from the routine. The wrapper functions are interposed between each caller and the called routine.

In general, in performing the monitoring experiments and constructing shared libraries, we have been impressed with the utility of the module operations for manipulating symbolic interfaces. We have found ourselves using these operations to “tweak” object files in both simple and complex ways, where previously we would have edited a new copy of the C source file, or tediously generated and modified assembly code. We can see that link-level facilities such as this, while not terribly glamorous, are badly needed when combining objects produced in different contexts.

7 OMOS Portability

Because OMOS is implemented in a portable fashion and makes use of relatively unsophisticated operating system functionality, it is highly portable.

As its first major step towards portability, OMOS does not rely on the availability of PIC in order to provide a shared library service. As mentioned, OMOS can make use of PIC if it is available, but we contend that we provide the same flexibility as PIC at no additional cost in the common case (i.e., few redundant copies of cached libraries), with an improvement in speed in the common case.

OMOS is written in C++, and C++ objects have been used to encapsulate important operating system functionality. As a result, only a small number of highly localized changes must be made in order to port OMOS to a new system. OMOS requires support for some form of on-demand mapping from files (e.g., mmap), some file system I/O capabilities, and the ability to allocate heap memory at particular locations.

Optionally, invocation of OMOS can be integrated into the operating system `exec` code. This is straightforward in modern Unix systems. The only required extension to the Unix interface itself, is some way efficiently to map memory into an unrelated process, which we do on Mach with `vm_map()`. A simple and easy to implement extension to the BSD or System VR4 `mmap()` interface would provide this: `pmmap(pid, ...)`, a privileged call.

Finally, OMOS requires an understanding of the native object file format. Although this understanding has also been encapsulated in an object, it remains the most complex and messy portion of the system to port. A promising route for future portability is the GNU project’s BFD[5] library. This library provides a machine and object-format independent interface to object files. It contains an array of object-format specific backends, and backends exist for most popular formats and machines. We are in the process of fitting this object file switch as a back-end to OMOS. Although a potential problem is inefficiency in speed and size, we expect this library will solve most of these portability issues.

Implications for Other Programs

There are a number of existing Unix utilities which read object files, such as debuggers, `nm`, `size`, and `strings`. Except for debuggers, each program is concerned with only a small part of the whole file, e.g., symbol table, string table, or `exec` header. They currently expect to see a simply structured sequence of bytes, from which they extract the few portions of interest. When such a program `open()`'s a file, it would make little sense for OMOS (integrated with the operating system, e.g. as a Virtual File System[10]) to pass it an entire byte stream, only to have the program discard the majority of the data. This is especially expensive and inappropriate when program segments are sparsely laid out in memory, which is beginning to occur on Unix systems.³ In cases where OMOS is not integrated into the OS, the existing utilities would not comprehend the new formats presented.

A solution to all of these issues lies in the BFD library, which is already used by portable versions of these utilities. We are adding a new "OMOS" backend which directly invokes the server, requesting only those portions of interest.

8 Results

8.1 Status

OMOS is in experimental use as an object server running on top of the Mach operating system on PA-RISC (OSF/1 server) and ix86 (BSD server) platforms, and on HP-UX on the PA-RISC platform. OMOS supports communication via Mach IPC, Sun RPC, and System V messages. On HP's PA-RISC object file format, `SOM`, and on `a.out` format files, OMOS supports the Jigsaw module operators, as well as a number of other useful graph operations. Conversion of OMOS to use the BFD object file switch is underway. OMOS has been used to conduct experiments in automatically generating locality-of-reference optimization in running systems[14]. The basic OMOS system comprises 17,600 lines of C++ code, and uses C++ pointer overloading to implement an automatic reference counting scheme for memory management.

We also have a non-server version of OMOS, called the Object File Editor (OFE). It offers a traditional command interface and manipulates files in the normal Unix file namespace. OFE has proven very useful for manipulating object files in a traditional environment.

8.2 Performance

Methodology

All timings were collected on an HP9000/730 which had 64 MB RAM and two SCSI-2 disks (HP-UX), and 32 MB RAM and one SCSI-2 disk (OSF/1-MK). The HP730 has a 67 Mhz PA-RISC 1.1 processor with a pagesize of 4KB. On HP-UX, we used HP-UX version 9.01, AT&T's C++ 3.0.1 ("cfront"), and GCC 2.3.3.u3 (a Utah distribution of 2.3.3 with additional optimizations for the PA). On Mach (OSF/1-MK), we used our HP700 ports of the Mach 3.0 kernel, version NMK13, and the OSF/1 single server, version 1.0.4b1 (derived from OSF/1 1.0.4). All files were on local disk.

Timings were obtained using the GNU `time` program on HP-UX and with the `csh`'s built in `time` command on OSF/1-MK. (Note that on Mach, the "system" cpu time is currently meaningless (low), since separate threads in the server provide most system services to the user program.) The systems were in multiuser mode, but idle, as verified by `vmstat`. Each run was repeated at least three times, with very little variance observed, less than 2%.

Two programs were studied. One was Unix `1s` (the 4.3 BSD version on HP-UX and the OSF/1 version on OSF/1). The other was `codegen`, part of the Alpha_1 geometric modeling system, a large system totaling 650,000 lines of C++ and 150,000 lines of Lisp. `Codegen` itself consists of 5,240 lines of code stored in 32 files. In addition, it relies on six libraries: two Alpha_1 libraries as well as `libm`, `libl`, `libC`, and `libc`. An

³This is already a problem with existing shared library systems— one frequently gets spuriously huge core dumps due to such memory gaps.

Table 1: Constraint-based Shared Library Performance Times in Seconds

HP-UX				
Test: <code>ls</code> (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	4.16	2.23	16.45	
OMOS bootstrap exec	1.63	14.57	16.56	1.007

HP-UX				
Test: <code>ls -laF</code> (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	121.8	169.4	294	
OMOS bootstrap exec	80.8	189.2	276	.93

HP-UX				
Test: <code>codegen</code> (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	211.1	75.9	289	
OMOS bootstrap exec	182.4	54.5	238	.82

Mach 3.0 with OSF/1 Server				
Test: <code>ls</code> (300 iterations)	User Time	System Time	Elapsed Time	Ratio
OSF/1 Shared Lib	.89	4.46	38	
OMOS bootstrap exec	1.50	5.62	23	.60
OMOS integrated exec	.89	4.49	17	.44

optimized, statically linked version of the program contains roughly 1,000 functions, with a total text size of 203KB and data size of 53KB. Tests were performed on a non-optimized, debuggable version with 289KB of text and 348KB of data. During timing the `codegen` program was run on a small input dataset which required reading three small files, and generated a single small file redirected to `/dev/null`.

Timings

In preliminary timing tests presented in Table 1, we compared the same implementation of `ls` constructed using HP-UX shared libraries⁴, and OMOS self-contained shared libraries, configured to use System V messages to communicate with OMOS. For a simple `ls` performing a list of a directory with a single entry, the OMOS and HP shared libraries performed similarly. (HP-UX must do some relocations OMOS does not, but the OMOS bootstrap program must do some IPC that HP-UX does not.) When we increase the number of system calls (by adding the `-laF` flags to the `ls` command), OMOS performance becomes proportionally better, exceeding HP-UX's by 7%. This was due to "user" time increases in the HP-UX shared library case. Since we used the default `-B deferred` binding mode, the HP-UX implementation[6, 15] does lazy relocation on procedure, and to some degree, data references, this time was presumably spent doing relocations and dispatch table patching. In timing tests using the much larger `codegen`, OMOS shared libraries were 18% faster than HP-UX shared libraries. Again, we see the effect of increasing numbers of relocations performed on every program invocation in the case of HP-UX, but only once for OMOS.

Both of the tested programs execute for relatively short times. On longer-running programs, the proportional speedup using OMOS would tend to be less, because in the traditional design, the majority of the

⁴ All HP-UX timings are reported using demand-loaded binaries. Our tests showed that demand-loading made no significant difference to the results.

relocations are presumably performed at startup. However, our implementation is also paying a significant startup cost, the IPC from the bootstrap loader to OMOS. This would tend to counteract the above effect.

On Mach 3.0-OSF/1, we compared our implementation and the OSF/1 1.0 scheme[1]. Since different compilers⁵ supported the two implementations of shared libraries, we could not ensure a completely valid shared library comparison, but we do see that the both the bootstrap and integrated `exec` versions of OMOS perform remarkably well, with the latter giving a 56% speedup. We are still in the process of measuring exactly where the gains are derived, but we believe that a shorter code path resulting from pre-parsing the executable file may be partially responsible. On tests made on the 386 version of Mach, OMOS integrated `exec` performed 33% faster than the native version, reinforcing this belief.

9 Related Work

A user-space loader is no longer unusual. Many operating systems, even those with monolithic kernels, now use an external process to do program loading involving shared libraries, and therefore linking. However, the loader/dynamic linker is typically instantiated anew for each program, making it too costly for it to support more general functionality such as in OMOS. Also, these loaders are not constructed in an extensible manner.

Other shared library designs give some degree of flexibility over symbol resolution. Functional substitution is usually supported, but without the fine control we offer— for example, it is difficult or impossible fully to control references from within shared libraries to a user-provided substituting or interposing function. It is common to support the runtime overriding of library search path rules by an environment variable (SunOS, System VR4, and HP-UX). Some versions of SunOS support the `LD_PRELOAD` environment variable which specifies the shared libraries to load before loading those requested by the application itself. In this way, some simple forms of runtime functional substitution can be provided. In addition, some linkers provide simple symbol renaming on the command line.

Packages exist, such as `dld`[9], to aid programmers in the dynamic loading of code and data. These packages tend to have a procedural point of view, provide lower-level functionality than OMOS, and do not offer the control over symbol manipulation that OMOS provides. `Dld` does offer dynamic unlinking of a module, which OMOS currently does not support. Since OMOS retains access to the symbol table and relocation information for loaded modules, unlinking support could be added. The CLAM system from the University of Wisconsin provides dynamic loading of C++ code in order to extend graphical user interfaces.

The Apollo DSEE[2] system was a server-based system which managed sources and objects, taking advantage of caching to avoid, typically, recompilation. DSEE was primarily a CASE tool and did not take part in the execution phase of program development.

10 Future Work

Many interesting problems remain to be addressed in this shared library scheme. We plan to develop policies whereby several instantiations of an OMOS meta-object, such as a shared library — each tuned for a different use — can be made available to client applications. Properly tuned policies would have beneficial effects on the working set sizes of programs using shared libraries.

The current constraint system uses primitive criteria for making decisions about object placement. A more sophisticated constraint system, based on the University of Washington's "Delta-Blue" constraint solver[17], has been developed in LISP and is being ported to OMOS and C++.

We are planning to extend the module operations to allow discrimination between symbol references and definitions. This will allow more flexible control over binding. For example, it will then be possible to better handle recursive functions.

There are many engineering issues to be addressed in OMOS: consolidating OMOS servers in a network,

⁵The implementation problem was the incompatible object formats each compiler generated, not the generated code itself.

refining the policies for managing main memory and backing store, and elaborating client interfaces. Using primitive fragments consisting of a single routine would gain flexibility, but it is unclear whether data structures can be made efficient enough for this to be feasible.

The extensible nature of OMOS, and its knowledge of everything from source file to execution traces, make it applicable to optimizations requiring run-time data. We are currently undertaking a project to exploit OMOS' specialization abilities to transparently optimize RPC. As another example, OMOS could transparently implement the type of monitoring done by MIPS' *pixie* system, to optimize branch prediction. Another direction is suggested by OMOS' natural connection with program development. OMOS could easily be used as the basis of a CASE tool, where its ability to feed back data from program execution would be useful for both debugging and optimization.

11 Conclusion

We have described a shared library design based on OMOS, a persistent server, providing fast and portable implementations of shared libraries, with superior flexibility both in symbol manipulation and implementation options. Preliminary performance results show it to be faster than some existing shared library schemes.

Acknowledgements

We are grateful to Peter Hoogenboom for providing OFE, BFD, and helping with other object format support. Thanks also go to Jeff Law and Mike Hibler for helping to integrate OMOS with OSF/1 *exec*, and to the other members of CSS, for providing their expertise in times of need.

References

- [1] Larry W. Allen, Harminder G. Singh, Kevin G. Wallace, and Melanie B. Weaver. Program loading in OSF/1. In *Proceedings of the Winter 1991 USENIX Conference*, pages 145–160. USENIX Association, Winter 1991.
- [2] Apollo Computer, Inc, Chelmsford, MA. *DOMAIN Software Engineering Environment (DSEE) Call Reference*, 1987.
- [3] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992. 143 pp.
- [4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20–23 1992. IEEE Computer Society.
- [5] Steve Chamberlain. *The Binary File Descriptor Library*. Cygnus Support, Palo Alto, CA, 1992. in FSF *binutils* distribution; Copyright Free Software Foundation.
- [6] Cary A. Coutant and Michelle A. Ruscetta. Shared libraries for HP-UX. *Hewlett-Packard Journal*, pages 46–53, June 1992.
- [7] Robert A. Gingell. Evolution of the SunOS programming environment. Unpublished report, Sun Microsystems, Inc., 1988.
- [8] Robert A. Gingell. Shared libraries. *Unix Review*, 7(8):56–66, August 1989.
- [9] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software— Practice and Experience*, 21(4):375–390, April 1991.

- [10] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference*, pages 238–247, Atlanta, GA, Summer 1986.
- [11] John Kohl and Carol Paxson. How well do SunOS shared libraries work? Unpublished report, Computer Science Division, University of California at Berkeley, December 1991.
- [12] Gary Lindstrom and Robert R. Kessler. Mach Shared Objects. In *Proceedings Software Technology Conference*, pages 279–280, Los Angeles, CA, April 1992. DARPA SISTO.
- [13] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 200–209, Paris, France, September 1992. IEEE Computer Society.
- [14] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993.
- [15] Marc Sabatella. Issues in shared libraries design. In *Proceedings of the Summer 1990 USENIX Conference*, pages 11–24, Anaheim, CA, Summer 1990. USENIX.
- [16] Marc Sabatella. Lazy evaluation of C++ static constructors. *ACM SIGPLAN Notices*, 27(6):29–36, June 1992.
- [17] M. Sannella, B. Freeman-Benson, J. Mahoney, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. Department of Computer Science and Engineering TR-92-07-05, University of Washington, 1992.
- [18] Donn Seeley. Shared libraries as objects. In *Proceedings of the Summer 1990 USENIX Conference*, pages 1–12, Anaheim, California, Summer 1990. Usenix Association.

Author Information

Douglas Orr is a Ph.D. student in the Computer Science Department at the University of Utah. His research interests include distributed systems, compilers and object systems. He is the primary architect of OMOS.

John Bonn is an M.S. student in Computer Science. His interests lie in operating systems and object oriented programming. Prior to joining the department John worked in industry for 5 years, specializing in real-time signal processing systems and Unix internals.

Jay Lepreau is Assistant Director of the Center for Software Science, a research group within Utah's Computer Science Department which works in many aspects of systems software. He has worked with Unix since 1979, and has served as co-chair of the 1984 USENIX conference and on numerous other USENIX program committees. His group has made significant contributions to the BSD and GNU software distributions. His current research interests include dynamic software system structuring for performance and flexibility, with operating system, language, linking, and runtime components.

Robert Mecklenburg is a Research Assistant Professor in the Computer Science Department. His major research interests are large scale software development, mixed language object-oriented programming, and persistence in object-oriented languages. He received his Ph.D. in Computer Science from the University of Utah in June, 1991. His dissertation, entitled *Towards a Language Independent Object System*, described a facility for mixing object-oriented programming languages with the goal of code reuse and new project development in multiple languages.

The author's addresses are: Center for Software Science, Department of Computer Science, University of Utah, 84112. They can be reached electronically at {dbo,bonn,lepreau,mecklen}@cs.utah.edu.

High Performance Dynamic Linking Through Caching

Michael N. Nelson Graham Hamilton

*Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043 USA*

Abstract

The Spring Operating System provides high performance dynamic linking of program images. Spring uses caching of both fixed-up program images and partially fixed-up shared libraries to make dynamic linking of program images efficient, to reduce the need for PIC (position-independent code), and to improve page sharing between different program images running the same libraries. The result is that with program image caching, dynamically-linked programs have a start-up cost similar to statically-linked programs regardless of the number of unresolved symbols in dynamically-linked program images and shared libraries. In addition, with library and program image caching, we have reduced the need for PIC and have increased page sharing.

1. Introduction

Dynamic linking of shared libraries is used in most modern UNIX systems. These systems include, among others, SunOS [2] and SVR4 [11]. Dynamic linking has several good properties:

- Unresolved symbols in a program do not have to be resolved until the program is launched. This decreases the time to build an image.
- Shared libraries do not have to be statically bound into a program image thus saving disk space.
- Shared libraries can be shared between many executing images thus reducing memory costs.
- Program images can take advantage of new releases of shared libraries without being rebuilt.

Unfortunately, by deferring the linking of programs to program start-up time, and possibly even run time, dynamically-linked programs start slower than statically-linked programs. This degradation was believed to be an acceptable cost to pay for the advantages of dynamic linking. However, with the advent of languages such as C++, the number of symbols that must be resolved at run time is dramatically increasing thus increasing the start-up time of dynamically-linked programs to unacceptable levels.

This paper describes the dynamic linking architecture and implementation in Spring, a new distributed object-oriented operating system. We had six main goals for the Spring dynamic linker:

- Substantially reduce the start-up time for dynamically-linked images.
- Maintain a high degree of page sharing between different images running the same libraries.
- Provide a dynamic linking solution that performs well even when the number of relocatable symbols increases dramatically.
- Run SunOS binaries on Spring.
- Support existing SunOS linkage semantics.
- Have non-PIC (position-independent code) libraries and main programs without significantly impacting program start-up time or page sharing.

The Spring dynamic linking system described in this paper achieves all six goals by caching fully linked program images and partially fixed-up shared libraries. After a dynamically-linked program is run once, future instances of the program will require no dynamic linking, and once a shared library is linked for one program, subsequent programs that use the same shared library will link much more quickly.

The rest of this paper is organized as follows: Section 2 provides an overview of the Spring Operating System; Section 3 presents related work; Section 4 describes the SunOS dynamic linking implementation; Section 5 describes the Spring dynamic linking implementation; Section 6 gives some performance measurements; Section 7 looks at the relevance of this work to other operating systems; Section 8 proposes future work; and Section 9 offers some conclusions.

2. Spring Operating System

Spring is a distributed, multi-threaded operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of an object and its operations is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the *server* (implementor) and the *client* of the object. Since Spring is object-oriented, it supports the notion of *interface inheritance*. An interface that accepts an object of type *foo* will also accept a subclass of *foo*.

A Spring domain is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the client of other objects. The server and the client can be in the same domain or in different domains. In the latter case, the representation of the object includes an unforgeable nucleus *door identifier* that identifies the server domain [3].

The Spring kernel supports basic cross-domain invocations, threads, and provides basic virtual memory support for memory mapping and physical memory management [6]. A typical Spring node runs several servers in addition to the kernel as shown in Figure 1. All services are exported via objects defined using the interface definition language. In general, any collection of servers may reside in the same or in different domains. The decision on where to run a particular server is made for administration, protection, and performance reasons, and is independent of the interface of the service.

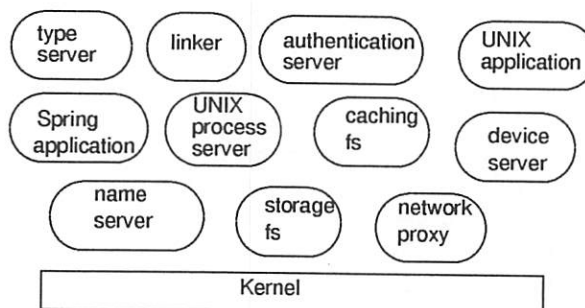


FIGURE 1. Major system components of a Spring node

2.1. UNIX Subsystem

Spring can run most dynamically-linked SunOS binaries using a UNIX subsystem [5]. This includes basic UNIX programs such as *cat* and *csh* and more sophisticated programs such as *openwindows* and most X applications. The UNIX subsystem consists of two parts: a UNIX process server that is responsible for managing process ids, ptys, sockets, and signals between processes, and a shared library *libue.so* that contains code to emulate all UNIX system calls. When a SunOS dynamically-linked binary is run on Spring, the *libc* shared library that is used to dynamically link the binary is actually a copy of *libue.so*.

3. Related Work

Dynamic linking was first used in the Multics [9] and TENEX [7] operating systems. However, it was not used in mainstream UNIX systems until System V [1] and SunOS [2] added dynamic linking support. More recently, SVR4 [11] and HP-UX [10] provided dynamic linking support similar to the SunOS dynamic linking architecture. We will describe the SunOS implementation in detail in Section 4.

The notion of caching previously run programs has been used in past UNIX systems. Early versions of UNIX had the notion of the “sticky” bit. If this bit was set, then the program would remain cached in memory after it exited. The next time that the program was run it would not have to be loaded into memory. However, to the best of our knowledge, this same strategy of caching previously run programs has not been used for dynamically-linked programs. Thus, every time that a dynamically-linked program is run it must be relinked from scratch.

4. SunOS Background

In this section we will examine dynamic linking on SunOS. We will first examine the notion of position-independent code and its effectiveness on C++ code, then we will examine how SunOS performs dynamic linking, and finally we will examine some libraries and programs to get some insight into what needs to be done to perform efficient dynamic linking on Spring.

4.1. PIC (Position-independent Code)

The Sun compilers support two flavors of output code: position-dependent code and position-independent code. Position-dependent code is the default. In position-dependent code the compiler makes no attempt to minimize the number of fix-ups that must be performed at dynamic link time. Typically one relocation will be required for each source code reference to an external symbol.

In PIC, the compiler and the static linker conspire to attempt to minimize the number of relocations that must be performed at dynamic link time by grouping as many external references as possible into tables, so that there is a single reference to each symbol. The code that uses these symbols then indirections (using PC-relative addressing) through these tables. For example, rather than performing a direct procedure call, PIC will load an address from an indirection table and jump through that address.

PIC has two advantages. First, it reduces the number of dynamic linkage relocations that need to be performed since a widely used procedure still only ends up with one table entry that needs to be relocated. Second, it also reduces the number of pages that will be updated by the relocations, since the table entries are clustered closely together. Reducing the number of page updates is very important as this maximizes the number of physical pages that can be shared between different users of the library.

4.2. PIC and C++

PIC successfully minimizes direct-instruction-stream references to relocatable symbols by adding a level of indirection. However, this approach cannot be applied to references from program data. If a given memory location is supposed to contain a pointer to a given relocatable symbol, then the compiler cannot simply add a level of indirection. This is not normally a significant problem for C programs as it is relatively rare for C data structures to be declared with pointers to procedures or other data.

The situation with program data references is rather different in C++. Both the cfront 3.0 preprocessor and the G++ compiler generate virtual function tables as initialized data structures. These virtual function tables are full of references to relocatable symbols. Thus, even when using PIC, a typical C++ program or library will contain a much higher number of symbolic relocations than a similar C program (see Section 4.4).

Although PIC reduces the number of symbolic resolutions over non-PIC, PIC does tend to be larger than non-PIC due to the addition of extra levels of indirection. With the G++ compiler, use of PIC increases the size of C++ texts between 3 and 20 percent, with roughly similar impacts on execution performance. The overall size of the core Spring library *libspring.so* that is written in C++ increases by 13 percent when compiled PIC with the G++ compiler. While this size increase is an acceptable price to pay for the benefits of shared dynamic libraries, it is not an insignificant cost. We were interested in determining if we could reap the benefits of shared dynamic libraries without paying this particular space and time tax.

Note that the increase in code size due to PIC is fairly sensitive to particular details of compiler technology and CPU architecture. Measurements of systems with other compilers may yield different results.

4.3. SunOS Dynamic Linking

In this section we will briefly describe the SunOS dynamic linker. For more details on the SunOS dynamic linker see [2].

Each *a.out* file contains as part of its text segment two tables used for dynamic linking. The first table is the symbol table. This contains a description of each symbol that is either defined or referenced by the *a.out* file. There is only a single entry for each symbol, even if it is both defined and referenced. The symbol table is indexed by an associated hash table that facilitates the mapping from a linker symbol to a particular symbol table entry.

The second table in each *a.out* file is the relocation table. This table contains a description of each location in the *a.out* file that needs to be patched up at dynamic link time. There are, broadly speaking, two kinds of relocations. The first kind does not require symbol resolution. This category basically covers relocations that are purely a matter of patching up code and data to reflect the addresses they were loaded at in memory. The second kind of relocations are symbolic relocations that are dependent on the final values of some particular linkage symbol. Each of these relocations contains a reference (actually a table index) to the symbol table entry describing the corresponding symbol.

A runnable image is composed from a set of *a.out* files, one of which is a main program and the rest of which are shared libraries. At dynamic link time, each of the relocations is processed in turn. Non-symbolic relocations are straightforward and can be performed solely on the basis of the address at which the *a.out* file has been mapped into memory. Symbolic relocations are more troublesome. For each symbolic relocation, the dynamic linker finds the symbol table entry associated with the relocation. This now gives the linker the name of the symbol that is being used for the relocation. It now searches through each of the *a.out* files in turn, searching for the first *a.out* file that contains a definition of the symbol. The *a.out* files are searched in a standard order, starting with the main program and then with the libraries in the order that they were specified on the static link command line, left to right. Thus for example, if one linked an image:

```
ld -o foo foo.o /lib/fred.so /lib/libc.so
```

then when resolving a symbolic relocation, *foo* would be searched first followed by *fred.so* and *libc.so*.

When a definition of the given symbol is found in a given *a.out* file, the linker combines the symbol table entry and the address at which the *a.out* file is loaded to calculate the absolute address to which the symbol refers. That value is then used to perform the relocation.

4.4. Looking at Libraries and Programs

Table 1 contains some statistics on several typical libraries. Some of these libraries are compiled PIC and the remainder are compiled non-PIC. The column labeled "defined symbols" is the number of symbols defined within the library, and the column labeled "undefined symbols" is the number of additional symbols that the library references but does not define. The table also includes the total number of relocations for each library, the number of these that

were symbolic relocations, and the number that were references to undefined symbols. The *liblinker* library is a

Shared Library	Source Language	Compiled PIC?	Defined symbols	Undefined symbols	Total relocations	Symbolic relocations	Undefined relocations
liblinker	C++	Yes	179	243	659	313	231
liblinker	C++	No	187	231	2699	1691	1587
libns	C++	Yes	498	416	1392	885	621
libns	C++	No	513	398	7497	5763	5287
libue	C++ and C	Yes	10162	5	10330	4980	5
libue	C++ and C	No	10175	5	41479	28168	5
libspring	C++	Yes	8772	0	7622	4073	0
libspring	C++	No	8780	0	38652	26948	0
libc	C	Yes	1048	6	1211	500	6
libc	C	No	1041	6	10424	4532	39

TABLE 1. Library Statistics

Spring library that contains the Spring linker implementation; the *libns* library is a Spring library that contains the generic Spring nameserver implementation; the *libue* library contains Spring UNIX subsystem code; *libspring* is the standard Spring library that is linked with all Spring applications; and *libc* is the standard SunOS library that is linked with SunOS applications.

Table 1 shows several interesting things. First, PIC is very good at eliminating relocations; it reduced them by a factor of five for the core Spring library *libspring*. Second, the Spring library *libspring*, which is written in C++, is much larger in terms of relocations and defined symbols than the SunOS *libc*, which is written in C; *libspring* compiled PIC contains six times as many relocations and eight times as many defined symbols as *libc* compiled PIC. Finally, although there are a large number of symbolic relocations for *libspring* when it is compiled non-PIC (over 26,000), they are all to symbols that are defined in *libspring* itself. Now some small number of these symbols may be redefined in other *a.out* files in a linked image, but this is comparatively rare. Almost all the relocations defined in *libspring* will end up being satisfied from *libspring* itself.

Table 2 contains statistics for some main programs. Note that for main programs all remaining relocations are symbolic relocations to undefined symbols. The top six programs are programs written in C and compiled for SunOS, and the bottom six are programs written in C++ and compiled for Spring. All libraries used by the programs are shared libraries.

Program	Source Language	Compiled PIC?	Defined Symbols	Undefined Symbols	Total Relocations
csh	C	Yes	722	83	83
csh	C	No	721	83	106
xterm	C	Yes	310	196	379
xterm	C	No	309	196	405
tar	C	Yes	112	66	66
tar	C	No	111	66	210
ssh	C++	Yes	109	321	819
ssh	C++	No	117	277	2015
caching_fs	C++	Yes	670	444	524
caching_fs	C++	No	696	416	4379
machine_ns	C++	Yes	21	126	81
machine_ns	C++	No	20	126	307

TABLE 2. Program Statistics

Table 2 shows two things. First, it once again shows the effectiveness of PIC in reducing relocations. Second, Spring programs have many more relocations than SunOS programs even when the Spring programs are compiled PIC. For example, *ssh* defines the same number of symbols as *tar* yet *ssh* has over ten times the relocations.

The results from the measurements of the Spring libraries and programs show that Spring has to deal with many more relocations than SunOS. Thus, Spring needs a dynamic linking solution that can efficiently link programs and libraries with large numbers of relocations.

4.5. Comments on the SunOS algorithm

Note that the non-symbolic relocations for a given *a.out* file are independent of which other *a.out* files are bound into the program image. These relocations only depend on the address at which the library is loaded. The main cost of dynamic linking is resolving the symbolic relocations and, in particular, in searching through the libraries for symbol definitions.

In typical small programs, almost all the definitions will come from the standard OS library (e.g., *libc* on UNIX or *libspring* on Spring), but for each symbolic relocation it is first necessary to search the symbol table of the main program and any other minor libraries. This property was somewhat mitigated in an earlier Spring dynamic linker [4] by the use of a cache of resolved symbols, but even so, it was by far the dominant component of the linkage.

For most programs, the dynamic libraries used by the program contain substantially more relocations and symbol definitions than the program itself. This is true both for conventional UNIX utilities such as *csh*, *awk* or *cpp*, and also for window-based tools such as *xterm* and *cmdtool*. Clearly there are also programs where the program contains more relocations and symbols than the libraries, but this is not the case for most of the programs for which fast start-up is important. Thus, we were particularly interested in minimizing the costs of fixing-up the libraries as opposed to fixing-up the program itself.

In order to reduce the cost of program start-up, the SunOS dynamic linker performs certain fix-ups only "on demand." This optimization is available for PIC by filling the indirection table for procedure references with pointers to a special fix-up routine. While this optimization is useful for SunOS, it is much less useful for Spring libraries as most of their relocations are data relocations, which cannot be performed lazily. We were therefore prepared to abandon this optimization, provided it did not cause any significant penalty during program start-up.

5. The Spring Dynamic Linker

In the last section, we showed that Spring shared libraries and programs have substantially more relocations than SunOS shared libraries and programs. For this reason, Spring needs a new dynamic linking implementation that will allow us to efficiently start dynamically-linked programs on Spring. In this section, we will describe the Spring dynamic linking mechanism that uses caching to provide excellent dynamic linking performance.

5.1. Spring Linking Model

Spring uses a different dynamic linking model than SunOS. In SunOS, a program image dynamically links itself as it begins executing. This is done through start-up code that is statically linked into each SunOS binary. In Spring, program images are linked by a parent domain and then the fully linked image is started. This model allows us to do the caching of linked images and libraries.

The Spring model has the disadvantage that the entire image must be linked at once, rather than on demand, as in SunOS. However, because of the effectiveness of caching in Spring, we pay very little penalty for our policy.

The actual linking of program images is done by a special linker domain described in Section 5.6. This domain not only links images but it is also responsible for caching linked images and libraries.

5.2. Motivation

An early version of the Spring dynamic linker is described in [4]. The new Spring dynamic linker described in this section follows the same basic linking model as the old Spring dynamic linker. However, the old dynamic linker had serious performance problems because of the large number of symbols that it had to relocate. When the original linker was built, there were over 50,000 relocations in the main Spring library *libspring* even when it was compiled PIC. Our first attempt at improving performance was to reduce the number of relocations. Through many different techniques, we have managed to drastically reduce the number of relocations in *libspring* to less than 8,000 when compiled PIC. Unfortunately, the original dynamic linker still did not have reasonable performance and the solution would not scale.

Our second attempt at improving performance was to use the caching mechanism described in this section. We wanted to have a solution that provided good performance and would scale if we decided not to use PIC or we once again found ourselves in the situation where we had a huge number of relocations.

5.3. Spring Dynamic Linker

The key feature of the Spring dynamic linker design is that it facilitates caching at several different levels. Rather than dynamically linking every image individually, we attempt to cache entire fixed-up images. When we miss in the image cache, rather than linking the entire image from scratch, we attempt to reuse existing cached libraries that are already provisionally fixed-up.

The linker domain has a range of addresses that it regards as suitable for dynamic libraries. Whenever the linker is asked to operate on a new dynamic library, it allocates the library space within this range at a location that will not conflict with any currently cached library. It will later arrange that the library is indeed loaded at this location in all dynamically-linked images.

Once the linker has allocated an address range for a library, the linker then provisionally fixes-up the library to reside at this location. It does this by making a copy-on-write copy of the library and then provisionally processing the list of relocations. All the non-symbolic relocations are performed immediately. For each of the symbolic relocations, if the symbol is defined in the library, a provisional relocation is performed based on the library's own definition.

When the dynamic linker is asked to fix-up an image, it finds the transitive closure set of the libraries needed by that program. For each of these libraries it obtains a copy-on-write copy of a provisionally fixed-up version of the library. The dynamic linker then performs a final fix-up pass on the main program and the shared libraries. All the relocations in the main program will be to symbolic relocations to undefined symbols, as all other relocations are fixed-up when the main program is statically linked. For each such relocation, the linker searches through the libraries for the first suitable symbol definition and uses that symbol's address to perform the relocation.

Fix-ups within the libraries are more interesting. The dynamic linker has already fixed-up all the non-symbolic relocations in the library and has provisionally fixed-up as many symbolic relocations as possible. However, the linker must now validate the symbolic relocations and perform any remaining ones. For each library, the linker compares the library's symbol table against the symbol tables for the linked image and for each library that occurs earlier in the search order. If a symbol that is defined in the library is also defined elsewhere, the linker redefines the library's symbol table entry to say that the symbol has been overridden. The linker then scans through each library's relocation table looking for relocations using undefined or overridden symbols. If the relocation points to an undefined symbol, then the linker can perform a standard relocation using the newly found symbol definition. However, if the relocation points to an overridden symbol, the linker must first undo the effects of the provisional relocation before performing the final relocation. This undo step may be unnecessary on some architectures, but on others (including SPARC), it is required as part of the relocation information is stored in the relocated location itself.

5.4. Assumptions and Consequences

The Spring dynamic linking strategy has a couple of assumptions and some resulting consequences. First, it assumes that the typical program defines a relatively small number of symbols compared with the number of symbols defined and referenced in the standard libraries. Thus, it is assumed that it is significantly cheaper to do a validation pass over a library to check that no symbol definitions have been overridden than it is to perform a full scale dynamic link of that library. This assumption would not be true if both a main program and a library defined large numbers of symbols (making the validation expensive), but the library had only a small number of relocations (meaning that a conventional dynamic link would be cheap).

The second assumption of the Spring dynamic linking strategy is that it is fairly rare for a main program to redefine a symbol that is defined in a library. If it were common for programs to redefine large numbers of library symbols, then many of our provisional fix-ups would be erroneous and the cost of undoing and redoing them may be more expensive than a straightforward dynamic link. Additionally, the page containing the redone fix-up will no longer be shareable with other processes that use the same dynamic library.

Fortunately, for the bulk of the programs we have looked at, both of these assumptions appear to be true. It is clearly the case that these assumptions will not be true for all programs, but even in these cases, the performance is unlikely to be significantly worse than a straightforward dynamic link.

There is one problem with the Spring linking strategy. A commonly used option “-dc” of the SunOS static linker causes space to be allocated in the main program for undefined external data symbols. Although for the standard SunOS dynamic linker this is a useful optimization, for our dynamic linker it is not. However, it is not normally a problem, except for two symbols “_errno” and “_iob”, both of which tend to be referenced in libraries as well as in the main program. Even for these symbols there is no real problem for libraries that are compiled PIC, since the compiler and static linker ensure that there is only a single reference to each of these symbols in each PIC library. However we wanted to be able to build our principal libraries non-PIC so as to avoid the space and time penalty caused by PIC.

In a better world, we might relink all of the SunOS programs without the “-dc” option. However, one of the goals of the Spring UNIX subsystem is to run standard SunOS binaries unchanged. Thus, we decided to compile PIC those affected *libue* modules.

5.5. Paging Behavior

The Spring dynamic linking strategy relies heavily on cheap copying of virtual memory segments using a copy-on-write virtual memory implementation where the physical pages are initially shared between the “copies” and are only really copied when a write is performed on one of the two memory segments.

A library is initially copied from a disk file to a provisionally fixed-up library and is then copied a second time to become part of a fully fixed-up image. Normally only these two copies are necessary, but if the linker image is being debugged a third copy will occur to generate a writable program image. Additional copies may occur due to UNIX emulation forks. Fortunately, the Spring VM system has no problems coping with multiple levels of copy-on-write sharing [6].

Both Spring and SunOS attempt to share dynamic library pages between different images. However, there is one key difference. SunOS copies the library first and then performs the fix-ups independently in each process that uses the library. This means that it is important to minimize the number of pages that are mutated by fix-ups, as these pages will not be shared.

The Spring dynamic linker, on the other hand, tries to perform most fix-ups early and then copies the fixed-up library. This is possible because all users of the library will load it at the same virtual memory address. Now, if Spring has to undo and redo its fix-ups, then we gain nothing. However, if the provisional fix-ups prove durable, then we

have potentially increased the number of physical pages that can be shared between users of the library. At the very least, Spring's strategy of caching entire fixed-up images will increase the amount of sharing between two processes running the same image, as they can both benefit from the same fixed-up pages.

5.6. Spring Linker Domain

The actual dynamic linking of images is done by a separate linker domain. This domain implements a *linker* object that it exports in a well-known place in the Spring naming service. When a new program is to be run on Spring, the parent domain invokes the *link* operation on a linker object. The arguments to the *link* operation include a memory object that represents the program image and a naming context to use to find shared libraries; this naming context is analogous to the loader library path in SunOS. After the linker domain links the image, it returns a list of <memory object, address> pairs. The parent domain then maps these memory objects at the given addresses in a new domain and starts the new domain executing.

The linker domain implements the caching of fully linked programs and partially fixed-up libraries. When the linker domain is asked to link a program it checks its image cache. If the memory_object that is being linked is the same as a previously linked image and the context that is being used to find libraries is the same, then the linker domain will consider this a cache hit. The linker can determine if two memory objects or contexts are equivalent by following a secure object equivalence protocol. This protocol requires two cross-domain calls for each equivalence check.

The linker domain strategy for determining if there is a hit on the image cache will be correct as long as the contents of the context used to find libraries does not change. If new libraries are bound into the context used for linking, then the linker may get a false cache hit. If new libraries are bound into the name space then the linker cache must be flushed to take advantage of these libraries. Since installing new shared libraries should be a fairly uncommon occurrence, we believe that our solution is reasonable. However, in order to eliminate the need for linker cache flushing, we plan to implement the more aggressive solution to cache coherency described in Section 8.3.

In addition to the image cache, the linker domain also maintains a cache of partially fixed-up libraries. When the image cache is missed, the image must be linked against a set of shared libraries. For each shared library, the linker domain uses the context given to the *link* operation to resolve the name of the library to a memory object. The linker then checks to see if the memory object is equivalent to the memory object for any cached library. If it is then the linker can use the already cached library.

6. Performance

This Section provides measurements of dynamic linking performance on Spring and SunOS 4.1.3. These measurements were taken on a SPARCstation 10 model 31 with a 36 Mhz CPU and 64 Megabytes of memory. The SunOS numbers include all of the linking that occurs before the main function is called. Thus, they include some fix-ups and the library mapping costs but do not include any on-demand fix-ups. The Spring numbers include the cost of fully fixing-up the shared libraries and the main program, but they do not include the library mapping cost. Since the mapping cost on SunOS and Spring is small, these mapping costs are not significant.

6.1. Linking Against a Single Library

Table 3 shows the time it takes to link a null main program on Spring and SunOS, and Table 4 shows the cost of linking some Spring programs on Spring and SunOS. These measurements show several things. First, image caching is very effective in providing low dynamic linking cost. If a program image is cached, then Spring can start-up the program in 12 milliseconds regardless of the number of relocations. However, SunOS, which does not have image caching, takes from 22 to 1376 milliseconds to link a program depending on the number of relocations in the library.

These measurements also show the effectiveness of library caching. Library caching allows the program to be linked up to four times faster than without caching. In fact, with library caching, the effect of a large number of relocations on linking performance is reduced. For example, even though a non-PIC version of *libspring* has five times as many relocations as a PIC version, it only takes 20 to 50 percent longer to link against a cached non-PIC version of the library. This is much better than SunOS which takes over five times longer to link against a non-PIC version of *libspring* than it does to link against the PIC version.

Library	Library Compiled PIC?	SunOS	Spring Program Cached	Spring Library Cached	Spring Nothing Cached
libspring	Yes	182 ms	12 ms	90 ms	185 ms
libspring	No	1008 ms	12 ms	136 ms	576 ms
libc	Yes	22 ms	12 ms	65 ms	73 ms
libc	No	197 ms	12 ms	87 ms	235 ms

TABLE 3. Linking A NULL Program

In general, even without any caching, Spring is able to link programs as fast or faster than SunOS. In fact when the number of relocatable symbols gets large, Spring can dynamically link twice as fast as SunOS even when neither the image nor the library is cached. Spring's better performance is due to the fact that the Spring dynamic linker caches and reuses symbol resolutions during the fix-up phase.

There is one case where without caching Spring is much slower than SunOS: when the null-program is linked against a PIC version of *libc*. The poor performance on Spring is because Spring has to perform all relocations before the program is run and SunOS does not. When libraries such as *libc* are compiled PIC, the SunOS idea of performing fix-ups of procedure references on demand appears to be very effective. However, for libraries such as *libspring* where most relocations are data relocations that cannot be performed lazily, the SunOS optimization is not very effective.

Program	Program PIC?	libspring PIC?	SunOS	Program Cached	Library Cached	Nothing Cached
ssh	Yes	Yes	209 ms	12 ms	134 ms	221 ms
ssh	Yes	No	1147 ms	12 ms	166 ms	588 ms
ssh	No	Yes	291 ms	12 ms	174 ms	272 ms
ssh	No	No	1286 ms	12 ms	212 ms	643 ms
caching_fs	Yes	Yes	239 ms	12 ms	158 ms	252 ms
caching_fs	Yes	No	1212 ms	12 ms	194 ms	615 ms
caching_fs	No	Yes	371 ms	12 ms	240 ms	335 ms
caching_fs	No	No	1376 ms	12 ms	280 ms	712 ms

TABLE 4. Linking Spring Programs on Spring

These measurements provide a good indication of the effectiveness of PIC. Without caching, PIC is very effective in reducing program start-up time. Compiling a library PIC is much more important than compiling a program PIC. The difference in the dynamic linking cost for PIC and non-PIC versions of a program is at most 50 percent. However, compiling a library PIC can reduce the dynamic linking cost by from three to eight times over non-PIC.

With image caching, PIC is completely unnecessary. In addition, library caching reduces the necessity for PIC. Thus, the need for PIC will depend on a combination of the hit rate of the image cache and the hit rate of the library cache. Our experience with Spring so far is that because of the effectiveness of our image and library caches, we have no need for PIC.

6.2. Multiple Libraries

The previous measurements were of the cost of dynamically linking a program against one library. Table 5 shows the cost of dynamically linking a null program against the set of X libraries *libXaw*, *libXmu*, *libXt*, and *libX11* and either *libc* or *libue*. Table 5 shows several interesting things. First, as expected, both image caching and PIC are very effective in improving performance. Second, without image caching, Spring is much worse than SunOS when the program and the X libraries are linked against *libc*. Once again, the poor performance on Spring is because Spring has to perform all relocations before the program can run and SunOS does not. Since each relocation must be checked against many libraries, the dynamic linking cost is very high (much higher than the cost shown in Table 3 of linking a null program against *libc*).

Library	Library Compiled PIC?	SunOS	Spring Program Cached	Spring Library Cached	Spring Nothing Cached
libue	Yes	486 ms	13 ms	424 ms	645 ms
libue	No	3111 ms	13 ms	473 ms	1017 ms
libc	Yes	102 ms	14 ms	416 ms	516 ms
libc	No	387 ms	13 ms	431 ms	684 ms

TABLE 5. Linking Against Multiple X Libraries

These measurements show the potential for the library chaining optimization described in Section 8.1 and the unified symbol table idea described in Section 8.2. We hope these optimizations would allow Spring to perform equally well regardless of the number of libraries that are being dynamically linked.

7. Relevance to Other Operating Systems

In Spring, it was natural to implement the dynamic linker cache as a user-level service. Spring's microkernel organization meant that image start-up was already handled by user-level libraries, and our focus on objects meant that system resources, such as memory objects, could be conveniently passed to and from a user-level service.

However the basic caching techniques we have described do not require a user-level server. A UNIX system might implement an equivalent dynamic library cache as an operating system service that is used by the *exec* system call. This should provide essentially all the same benefits as our user-level service.

8. Future Work

Everything described in this paper has been implemented. In this section, we will describe three potential improvements to our linking system.

8.1. Chaining

Some programs are linked with many shared libraries. For example, the X programs such as *xterm* and *xclock* are linked with five shared libraries. When a program with many shared libraries is linked in Spring, the linker domain will fix-up each library in turn each time different programs that use the same set of libraries are linked. One optimization that we could implement is to cache fixed-up chains of libraries. Thus, if two programs are linked that use the same set of libraries in the same order, then the second program could use the cached set of libraries that were fixed-up for the first program. As an example of the chaining optimization, consider the two programs *progA* and *progB* created in the following manner:

```
ld -o progA progA.o X.so Y.so Z.so
ld -o progB progB.o X.so Y.so Z.so
```

If *progA* is linked, the linker domain could cache the fixed-up chain of libraries $\langle X.so, Y.so, Z.so \rangle$. In this cached chain, *X.so* will already be fixed-up against *Y.so* and *Z.so*, and *Y.so* will already be fixed-up against *Z.so*. When *progB* is linked, the linker domain will notice that it uses the already cached chain of libraries $\langle X.so, Y.so, Z.so \rangle$. Thus when *progB* is linked many of the linking steps can be avoided.

8.2. A Unified Symbol Table

Currently, we maintain a separate symbol table for each library in our cache. However, a possible alternative would be to maintain a unified symbol table for all the libraries in the cache. This unified symbol table would map each symbol to the set of libraries that implement it (in practice the vast majority of symbols are only defined by a single library). Whenever we added a library to the library cache, we would add its external symbol definitions to the unified symbol table.

This unified symbol table could potentially accelerate the final linking stage for images that use large numbers of shared libraries. Currently, we have to search through each of the libraries' symbol tables in turn to resolve undefined symbols and to validate that a given symbol definition has not been overridden. By doing a single lookup in the unified symbol table, we would be able to discover which libraries define that symbol and determine relatively quickly which (if any) of those library definitions to use for the current image.

8.3. Aggressive Coherency

Our current mechanism, described in Section 5.6, for determining a hit in the image cache requires cache flushes when libraries change. We plan to be much more aggressive about keeping the image and library caches coherent with respect to changes in libraries. We intend to implement a fully coherent linker cache by using a combination of our naming and file caching strategies [8].

9. Conclusion

The Spring dynamic linker uses caching very effectively to improve performance. When a dynamically-linked program image is cached, then the program can start up nearly as fast as a statically-linked program. The only difference is that there are more memory objects to map for dynamically-linked programs. When libraries are cached, uncached programs can still start up quickly regardless of the number of relocations in a library. In fact, in the cases that we have measured on Spring, PIC is not necessary for good dynamic linking performance.

Current UNIX dynamic linking implementations were designed for languages such as C where the use of PIC can reduce the dynamic linking costs to an acceptable level. However, object-oriented languages such as C++, with features such as virtual inheritance, produce images where technologies such as PIC are no longer sufficient to provide acceptable dynamic linking costs. In this paper, we have presented a new technology, caching of fixed-up program images and shared libraries, that provides efficient dynamic linking of programs written in these new object-oriented languages. We believe that new operating system techniques such as image and library caching, or new compiler techniques, are going to be required in the future in order to ensure acceptable dynamic linking performance.

Acknowledgments

We would like to thank Yousef Khalidi and Peter Madany for providing valuable feedback on this paper. We would also like to thank Yousef for originally suggesting this line of inquiry. Finally, we would like to acknowledge the efforts of all the Spring contributors whose work made this project a success.

References

- [1] Arnold, Q., "Shared Libraries on UNIX System V," *USENIX Summer Conference*, 1986, pp. 395-404.
- [2] Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S., "Shared Libraries in SunOS," *USENIX Summer Conference*, 1987, pp. 131-145.

- [3] Hamilton, G. and Kougiouris, K., "The Spring nucleus: A microkernel for objects," *USENIX Summer Conference*, June 1993.
- [4] Kempf, J. and Kessler, P. B., "Cross-Address Space Dynamic Linking," Sun Microsystems Laboratories, Technical Report TR-92-2.
- [5] Khalidi, Y. A. and Nelson, M. N., "An Implementation of UNIX on an Object-oriented Operating System," *USENIX Winter Conference*, January 1993.
- [6] Khalidi, Y. A. and Nelson, M. N., "The Spring Virtual Memory System," Sun Microsystems Laboratories, TR-93-9.
- [7] Murphy, D. L., "Storage Organization and Management in TENEX," *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972, pp. 23-32.
- [8] Nelson, M. N., Hamilton, G., and Khalidi, Y. A., "A Framework for Caching in an Object Oriented System," Submitted for publication.
- [9] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [10] Sabatella, M., "Issues in Shared Library Design," *USENIX Summer Conference*, 1990, pp. 11-23.
- [11] *System V Application Binary Interface*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Author Information

Michael N. Nelson is currently a Senior Staff Engineer at Sun Microsystems Laboratories. Before joining Sun he was one of the principal developers of the Sprite Operating System at Berkeley and worked at DEC Western Research Laboratory. His interests include distributed, object-oriented software, operating systems, and architecture. He has a Ph.D. in Computer Science from UC Berkeley. He can be reached at Sun Microsystems Laboratories, Inc., 2550 Garcia Ave., MTV 29-112, Mountain View, CA, 94043, USA, or via e-mail at michael.nelson@sun.com.

Graham Hamilton is a Senior Staff Engineer at Sun Microsystems Laboratories, where he is the project lead for the Spring operating system project. His interests include distributed computing, object-oriented systems, and operating systems. He has a Ph.D. in Computer Science from the University of Cambridge.

Trademarks

Sun, Sun Microsystems, SunOS, and SPARCstation are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories, Inc. All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc. All other product names mentioned herein are the trademarks of their respective owners.

The Shell as a Service

Glenn Fowler

gsf@research.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

This paper explores the design history of the *nmake* shell coprocess. Originally a special purpose uniprocessor executor, the *coshell* has evolved into a general purpose service that automatically executes shell actions on lightly loaded hosts in a local network. A major thrust of this work has been ease of use. The only privilege required for installation, administration or use is *rsh* access to the local hosts.

nmake and *GNU-make* users can take advantage of network execution with no makefile modifications. Shell level access is similar to but more efficient than *rsh* and allows host expression matching to replace the explicit host name argument. Also provided is a C programming library interface with five primitive operations that follow the *fork-exec-wait* process model.

Beside the speedups attained by parallelizing computations in a homogeneous network, *coshell* also supports heterogeneous configurations. This presents novel solutions to traditional cross-compilation problems. It also allows the user to view a new network host as a compute engine rather than yet another architecture on which to port the home environment and tools.

coshell runs on most S5R4 and BSD UNIX* operating system variants.

1. Introduction

make [Fe179][SUN87][SV84] is a natural candidate for parallelization. It generates a directed graph of file dependencies with nodes as files and edges as dependency relationships. Edge direction points from *target* nodes to *prerequisite* nodes. Target nodes are optionally labeled with shell script actions. If any prerequisite is newer than its target then the target action may be executed by a shell [Bour78][BK89] to bring the target up to date. *make*'s job is to traverse the graph from a given root node and execute the actions for all targets that are out of date.

If the dependency graph specified by the input makefile is complete (no omitted dependencies) then the shell actions for some nodes can be executed in parallel. In the simplest case sibling prerequisites of the same target may be made concurrently:

```
a : b c d
```

Here the actions for *b*, *c* and *d* may execute concurrently, but the action for *a* must wait until its prerequisite actions *b*, *c* and *d* complete.

There is a catch in parallelizing old makefiles: because *make* implementation details have crept into the user interface, some old makefiles rely on the implied ordering that *b* is made before *c* and *c* is made before *d*. Parallel *make* implementations take one of two approaches to compensate. The first promotes the implied ordering to a feature that must be overridden by explicitly listing prerequisites that can be parallelized:

* UNIX is a registered trademark of USL.


```
a :& b c d
```

The second breaks the old implicit model and lets the user discover missing prerequisites by trial and error, as in:

```
y.tab.c : a.y
b.o : y.tab.h
a : y.tab.o b.o
```

Here `a.y` generates `y.tab.c` and `y.tab.h`, `b.c` includes `y.tab.h` and generates `b.o`. If the `b.o` action executes before `y.tab.o` then the compilation will fail because `y.tab.h` will not have been generated yet (or worse, the compilation will succeed by using an old and possibly incompatible `y.tab.h`).

A fundamental feature of *nmake* [Fowl85][Fowl90], *GNU-make* [SM89] and *mk* [Hume87] is that prerequisite lists are not ordered. In contrast to other makes, *nmake* also provides automatic `#include` prerequisite analysis that prevents errors of omission as in the previous makefile example. Because of this all *nmake* makefiles are by default parallelizable.

Handling the details of concurrent execution is only the first step towards network execution. A study of how various *make* implementations execute actions illustrates the next step.

2. Uniprocessor Implementation

The original *make* model divides each action into lines that are sent off, in order, to a separate shell invocation (`sh -e -c "line"`). This is why shell constructs like

```
a : b
    cd c; if test -d d; echo ok d; fi
    cd c; doit
```

are common in old makefiles. Besides complicating action syntax and semantics (why should an action be that much different from a shell script?), sending each line to a separate shell is inefficient. A simple command like `cc -e -c t.c` requires a fork and `exec` for the `sh -e -c` and another fork and `exec` by the shell for the `cc` (although some shell implementations like *ksh* optimize out the second fork). Most *make* implementations avoid the double fork by checking each line for shell metacharacters `*?;[]()'"`$\\` and skipping the shell `exec` when no metacharacters are present. Although the shell may be bypassed *make* must still duplicate the shell `PATH` command search rules. And, unless *make* becomes the shell itself, the `exec` optimization ignores shell functions and aliases.

Why not drop the line-by-line semantics and send entire actions to the shell? The counter reasoning lies in the old *make* execution model. First, old *make* allows action lines to be prefixed by the special characters `@` and `-`, where `@` means *do not trace this line* and `-` means *ignore command errors from this line*. If actions were not split into lines then *make* would have to do a shell parse to determine if an occurrence of `@` or `-` were special. Second, some actions taken as a unit would simply be too big to send to the shell via `sh -e -c`. Finally, as with parallelization above, *make* is nailed by a backwards compatible implementation detail: the previous example fails when executed as a unit.

To work around this inefficiency some implementations offer a `.MULTILINE` phony target for selective multi-line execution. Others, like *mk* and *nmake* execute actions as a unit by default and force makefile conversion for old makefiles (in practice line-at-a-time execution has not been a conversion problem).

Executing actions as a unit is not a prerequisite for parallelization, but it does provide a convenient abstraction for *make* actions as jobs. Another benefit is that it eliminates the need for extra `;` and `\` characters to force shell actions into old *make* syntax. Actions also form a natural, localized grouping for network execution.

The size limitation of passing an action as an argument to `sh -e -c` can be avoided by piping the action to the shell instead. For example, the following two command lines are equivalent:

```
sh -e -c '{ echo hi; echo lo; }'
echo '{ echo hi; echo lo; }' | sh -e
```

Connecting to the shell by a pipe not only removes action size limitations, it also makes it possible for a single shell to execute a sequence of actions:

```
{
echo 'action1'
echo 'action2'
} | sh -e
```

Up to this point all semantics of the original *make* model, except for line-at-a-time execution, have been preserved. This includes entities passed across *exec*, such as the environment and open file descriptors. Since *make* typically does not change these between action executions, it is possible to eliminate a shell *fork/exec* for each action by using a single shell to handle all actions. Using one shell requires another pipe to allow execution after failure. This pipe provides *status* information from the shell back to the caller. The action encapsulation, written to the shell *command* pipe, becomes:

```
{ action; } && echo 0 >&status || echo $? >&status
```

Here *status* is the shell side file descriptor of the status pipe. It is interesting to note that this encapsulation transforms *fork/exec* into a write on the shell *command* pipe and *wait* into a read on the *status* pipe (unintentionally similar to the original UNIX system *wait* implementation).

A final detail completes the sequential execution encapsulation: if *action* contains shell syntax errors then the shell will either exit or hang (e.g., waiting for " or fi on the *command* pipe). *eval* fixes this:

```
eval 'action' && echo 0 >&status || echo $? >&status
```

As for concurrent execution the shell side is easy -- just use "&". The caller, however, must maintain a table that associates a job id with each outstanding action so that the exit status messages can be identified:

```
{ eval 'action' && echo jobid 0 >&status || echo jobid $? >&status; }&
```

No problems are evident at first glance, but the *coshell* design process has been an ongoing education in shell internals. Lesson number one: shells with job control disabled (non-interactive shells like the one above) may not reap (wait for) child processes. This means that given enough & commands, even though all may have completed, the shell may exceed its process limit. So the *coshell* encapsulation must manually track action pids and execute *wait* commands to reap zombies. Lesson number two: the shell \$\$ variable (process id) is set once at shell startup and is not reset when the shell forks (i.e., it is *not* the pid of the current process). Taking this into account results in:

```
{ eval 'action' && echo x jobid 0 >&status || echo x jobid $? >&status; }&
echo j jobid $! >&status
```

This encapsulation has two *status* messages: the j message associates a process id with a *jobid* and the x message associates an exit status with a *jobid*. After the x *status* message is received the caller can send a *wait pid* command to the shell to reap the terminated action process.

A side effect of concurrency is that only one job (the *foreground* job) may read from the terminal at any one time. For job control shells the foreground job is manually determined. Rather than complicate the user interface and implementation, *coshell* disables interactive input for each action by redirecting the standard input from /dev/null. Any interactive actions must explicitly redirect input from the terminal by name, e.g., </dev/ttyxnn. This has not been an inconvenience in practice. Figure 1 illustrates the concurrent execution model.

3. Library Interface

The *coshell* library interface, declared in <coshell.h>, completes the concurrent execution model and hides action encapsulation issues from the user. It also makes *coshell* a publicly available resource rather than just another special purpose *make* hack. The interface routines are:

- *Coshell_t** *coopen*(char* shell, int flags, char* attributes): This function opens a connection to the coshell and returns a coshell handle to be used in the remaining library calls. The handle

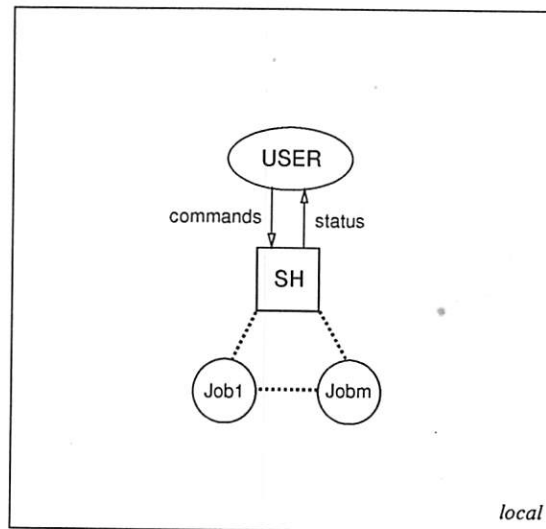


Figure 1. Concurrent execution model

points to a structure of readonly coshell accounting data. `shell` is the coshell process name and defaults to `ksh`. The environment variable `COSHELL` overrides the default value. `flags` controls job execution: `CO_SILENT` turns off job command tracing, `CO_IGNORE` ignores command errors within each job, and `CO_LOCAL` disables remote job execution. `attributes` is a string that is interpreted by the coshell and is ignored by all but the remote coshell described in the next section. If `attributes` is `NULL` then the value of the environment variable `COATTRIBUTES` provides a default value.

The readonly accounting data in `Coshell_t` is:

- `int outstanding`: The number of jobs that can be waited for by `cwait()`. Some of these may have already completed execution.
- `int running`: The number of jobs currently executing in the coshell. `Coshell_t.outstanding - Coshell_t.running` is the number of jobs that have completed execution but have not been reaped by `cwait()`.
- `int total`: The total number of jobs sent to `coexec()`.
- `unsigned long user`: The total user time in `1 / CO_QUANT` second increments for all jobs that have been reaped by `cwait()`.
- `unsigned long sys`: The total system time in `1 / CO_QUANT` second increments for all jobs that have been reaped by `cwait()`.
- `Cojob_t* coexec(Coshell_t* sh, char* cmd, int flags, char* out, char* err, char* att)`: This sends the shell command `cmd` to the coshell for execution and returns a job handle that points to a structure of readonly job accounting data. `flags` are the same as in `coopen()` and are used to augment the default settings from `coopen()`. `out` is the standard output file name and defaults to `stdout` if `NULL`. `err` is the standard error file name and defaults to `stderr` if `NULL`. `att`, if non-`NULL`, contains job attributes that are appended to the attributes from `coopen()` before being sent to the coshell. Job status may be checked after a call to `cwait()`. Application specific data can be attached to the `Cojob_t` user modifiable field `void* local`. The same `Cojob_t` handle is returned by `cwait()` and the `local` field is retained for additional user access.
- `Cojob_t* cwait(Coshell_t* sh, Cojob_t* job)`: This returns the status for `job`, or for the next job that completes if `job` is `NULL`. `cwait()` blocks until the specified job(s) complete; `NULL` is

returned when all jobs have completed. (sh->outstanding - sh->running) is the number of jobs that may be waited for without blocking. job->status is the shell exit code for the job. The return value data is valid until the next coexec(), cwait() or coclose().

- int cokill(Coshell_t* sh, Cojob_t* job, int sig): This sends the signal sig to job in the coshell sh. If job is NULL then the signal is sent to all jobs in the coshell.
- int coclose(Coshell_t* sh): This closes the connection to sh and returns the coshell exit status if available.

4. Network Implementation

Most UNIX systems provide shell level remote execution via *rsh* or *remsh*. Both require an explicit host name argument: *rsh host action*. A network *coshell* could be implemented by adding *rsh* to the action encapsulation:

```
{ rsh host eval 'action' && echo x jobid 0 >&status || echo x jobid $? >&status; }&
```

where *host* would be selected by coexec() to be the "best" host on the local network. This has the advantage that actual remote execution is deferred to a widely available implementation that requires no special privileges. On the downside:

- each action incurs the overhead of *rsh* setup
- coexec() must somehow implement a "best host" ranking
- an *rsh* design flaw that fails to report the remote action exit status complicates error detection
- *rsh* executes in the caller's remote home directory with a minimal default environment

These counter arguments lead the way to a different network *coshell* implementation. The key point is illustrated by the concurrent execution model in figure 1. Since all communication between the *coshell* and caller is by the *command* and *status* pipes, one could replace the *sh* process with one that acts as if it were *sh* and the caller would be none the wiser. Figure 2 illustrates this model.

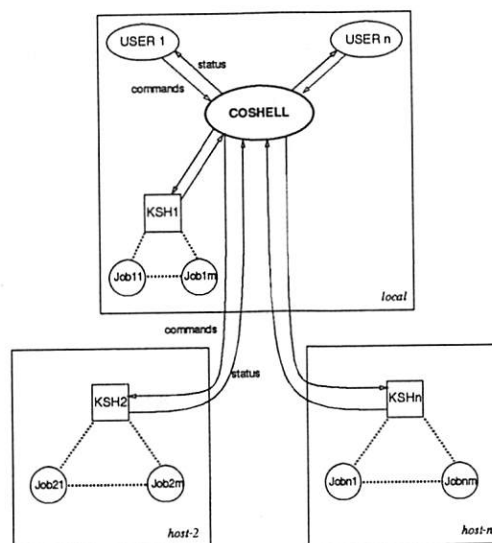


Figure 2. Network execution model

In this model the shell processes are distributed to hosts on the local network and a special *coshell* process sends user jobs to these shells. Instead of requiring privileged execution servers to be running on each host as some remote execution systems do [AC92][HP90], *coshell* uses *rsh* to establish a single shell process on each candidate host. And, in order to minimize the overhead of establishing these network shell connections, the network

coshell process is implemented as a daemon. Security is maintained by making the *coshell* daemon a per-user process. The effect is that a network *coshell* user has no more or less privilege than that provided by *rsh*. From an efficiency viewpoint *rsh* is only used to establish remote shell processes. Once the shell is running the *rsh* process exits, so that in the steady state a *coshell* daemon consumes one process for itself and one process for each active network shell connection.

The *coshell* daemon's job is to:

- Accept user connections via `coopen()`: daemon connections are based on the standard UNIX operating system networking IPC paradigms [LFJ83][Ritc84][Stev90]. One difference is that connections are initiated by the *coshell* process that is executed by `coopen()` rather than in the `coopen()` routine itself. This is a minor detail, but it means that there is no networking IPC code in the *coshell* library. In fact, network *coshell* was developed and tested without re-compiling or re-linking any of the *coshell* library based commands. Network execution was enabled by merely exporting `COSHELL=coshell` in the environment.
- Send `coexec()` jobs to the best hosts on the local network: determining the "best" host is the hardest component to implement efficiently. Since it involves scheduling it is also an area ripe with support from the literature [Bers85][DO91]. Each network shell connection is considered a resource (it takes time to establish a new connection) so old connections are kept active to minimize job overhead. Greed is counterproductive, however, as each shell connection accounts for a remote process slot and local file descriptor that must be polled. Not only is the "best host" of interest for assigning jobs to hosts, information on the top hosts is also necessary for optimizing the number of active network shell connections. As the host rankings change sub-par host connections are dropped to make room for better ones.

The literature tends towards centralized scheduling to provide an accurate basis for load balancing across the network. The problem is that a central scheduler becomes a communications bottleneck and potential security loophole: if the central scheduler also executes jobs then it must run as root and verify the identity of each request; if it only determines the best host then it must verify that the job is eventually executed on that host.

An alternative, used by *coshell*, is to use decentralized scheduling based on random smoothing. In this algorithm the hosts are ordered by a non-linear ranking function that is parameterized by the 1 minute load average, relative mips rating, and minimum idle time for all interactive users. The load average and mips rating give a measure of the capacity to execute new jobs whereas the minimum interactive idle time is used to make *coshell* a good citizen. After all, the goal should be to grab idle cycles rather than to sap your colleague's mouse and keyboard interactions. This is such a strong component of the ranking that by default no jobs are run on hosts with less than 15 minutes interactive idle time. A random component is added to the ranking to break ties. The effect is that unrelated *coshell* daemons will most likely have different rank orders, and, should both schedule jobs at the same time, different hosts will most likely be selected.

Host status is generated by a per-host status daemon supplied by the *coshell* library. Each status daemon posts the status to a host-specific file that is visible to all hosts on the local network. For efficiency the information is compressed into 8 bytes and stored in the access and modify time fields of the file status. `stat()` implements host status query and `utime()` implements host status update. Both of these operations on NFS requires at most 1 RPC -- the same efficiency that a hand-coded implementation could have attained, but with the added benefit of the file system abstraction.

Special system boot-time entries (e.g., `/etc/rc`) are not required. Instead the status daemon for each host is initiated by the first status request for that host. One daemon per host is maintained by keeping track of the `ctime` of the host-specific file. If the `ctime` has changed since the last status update then the daemon exits on the assumption that another daemon has taken over. This also provides a clean mechanism for killing status daemons: remove the corresponding host status file. The status update frequency is 40 seconds plus a random part of 10 seconds. The random component tends to evenly distribute the update times of all status daemons on the local network.

The *coshell* status daemon configuration scales linearly with the number of hosts as opposed to the widely used *rwho/ruptime* that uses an n-squared broadcast algorithm.

- Multiplex the standard output and standard error IO streams from the remote shells to the local user processes: this is the only *coshell* feature that requires a special shell extension. The standard output and standard error for each job must be redirected to network pipes that are intercepted by *coshell*. Data appearing on these pipes is then transferred to the appropriate file descriptors in the user processes on the originating host.

The extension allows connections to network pipes at the shell level. *ksh*, since the 88g release, has been modified to accept redirections to files of the form `/dev/tcp/n.n.n.n/port`. A redirection to such a file results in a connection request to the socket named by the host address *n.n.n.n* and port number *port*. *coshell* creates a `tcp` socket on the host address *n.n.n.n* and port *port* and listens for connections on the socket. The job encapsulation redirects standard output and standard error to this socket and initiates the *coshell* connection by sending a one line message on the connected socket that identifies the user process and file descriptor. This one simple extension makes it possible to use the shell for remote job execution rather than a special purpose process or daemon, as in *Remote UNIX* [Litz87].

- Retain most of the semantics of the concurrent execution model: user environment, processor type and remote file systems are the main issues. Most user environments contain at least 1K of data that seldom changes from the login values set by `.profile`. Passing the entire environment for each action would be prohibitive in communication and initialization overhead. *coshell* handles this situation by reading the user `.profile` and `$ENV` files to initialize remote shell connections. This increases the shell initialization overhead, and adds to the importance of active shell connection management. For the rare cases where some environment variables change between *coshell* invocations the environment variable `COEXPORT` is provided. Its value is a : separated list of environment variable names whose values are checked and exported by each `coexec()`. The `COATTRIBUTES`, `COEXPORT`, `FPATH`, `NPROC`, `PATH`, `PWD` and `VPATH` variables are always exported.

On the remote side the environment variables `HOSTNAME` and `HOSTTYPE` are automatically defined. As an aid to heterogeneous execution any occurrence of `/hosttype/` in the export variable values is changed to `/$HOSTTYPE/` and evaluated in the remote job context. For instance, if the local host type is `sun4` and `PATH=/home/bozo/arch/sun4/bin:/bin` then remote jobs will evaluate `PATH=/home/bozo/arch/$HOSTTYPE/bin:/bin`.

Host attributes are maintained in a central host description file:

local	busy=2m	pool=9	
bunting	type=sgi.mips	rating=60	
dodo	type=sun4	rating=9	idle=15m
gryphon	type=sun4	rating=18	cpu=3 os=solbourne
knot	type=sol.sun4	rating=4	idle=15m
lynx	type=hp.pa	rating=40	
parker	type=sun4	rating=21	idle=15m
quail	type=att.i386	rating=5	idle=15m
toucan	type=sun4	rating=22	cpu=4

Each line describes a single host; the first field is the host name followed by a list of *name=value* attributes. The attributes used by *coshell* are:

- **name:** the host name in the local domain (i.e., no `.`'s in the name).
- **type:** the host type that differentiates different processor types. Normally hosts with the same `.o` and `a.out` format have the same type.
- **rating:** the mips rating relative to the other hosts on the network. This is usually the observed rating rather than the one in the vendor's advertisements.
- **cpu:** the number of cpus for multi-processor hosts.
- **idle:** the minimum interactive idle time before jobs will be scheduled on the host. `idle` is usually 15m for workstations and is not specified (i.e., always available) for compute servers.

The special host name `local` defines default scheduling parameters for the *coshell* daemon:

- **busy**: a job running on a host that has become busy (non-idle) is allowed to continue running for this amount of time, after which it is stopped via SIGSTOP. The job is restarted via SIGCONT when the host once again becomes idle. Busy job status messages are posted to the tty where the *coshell* daemon was started, allowing the user to manually restart busy jobs if necessary. The reader may recognize this as a lazy alternative to process checkpointing and migration. Although far short of what a migration would provide, this method has the advantage that it preserves most process semantics (e.g., time is *not* preserved) and places no restrictions on processes that can run under *coshell* [BLL91]. In practice (mostly *nmake* users) with *busy=2m* this method has been activated only during tests to verify that it actually works.
- **pool**: a soft upper limit for the number of shell connections the daemon maintains. The limit may be exceeded to handle requests for host that otherwise would not have been added to the shell pool.

By default jobs are only scheduled on hosts that have the same **type** as the originating host. The environment variable *COATTRIBUTES* overrides this default by specifying a C-style host selection expression. For example, *COATTRIBUTES='(type=="sun4|hp" && rating>10.0)'*. User specified attributes of the form *name=value* may also be specified and queried, as in *COATTRIBUTES='(floating_point_accelerator==1)'*. Given the *HOSTTYPE* environment variable translation from above, cross-compilation with *nmake* is as simple as exporting *COSHELL=coshell* and *COATTRIBUTES='(type=="cross-compiler-type")'*.

The most important semantic to preserve is the file system. Any host used by *coshell* must be able to access the same files as the originating host. This requires administrative cooperation between the hosts but is not a problem in practice since most *coshell* hosts are workstations that share a small group of common file servers. Notice, however, that readonly files like those in */usr/include* need only follow the *as if* rule.

5. Command Interface

The *coshell* package includes 3 user level commands. *cs* (for connect stream) provides information on the active coshells and supports queries on the local host attributes. *cs -h* lists the host specific environment attributes for the local host. This is particularly useful for setting up *PATH* and *viewpaths* in *.profile*:

```
$ cs -h
HOSTNAME=dodo HOSTTYPE=sun4
$ eval $(cs -h)
$ print $HOSTNAME $HOSTTYPE
dodo sun4
```

cs -a attr lists host attribute information:

```
$ cs -a type local
sun4
$ cs -a - gryphon
type=sun4 rating=18 cpu=3 os=solbourne addr=135.3.113.106
```

ss (for system status) lists host system status generated by the daemon *ssd* and is similar to *uptime*:

```
$ ss
bluejay      up  2w00d,  1 user,  idle 2h56m, load 0.56, %usr 0, %sys 0
cardinal     up  2w00d,  1 user,  idle 19h33m, load 0.00, %usr 0, %sys 0
condor       up  2w00d,  1 user,  idle 34m00s, load 0.08, %usr 0, %sys 0
dgk          down 1d17h,  0 users, idle      0, load 0.00, %usr 0, %sys 0
dodo         up  2w00d,  4 users, idle 1.00s, load 0.32, %usr 0, %sys 2
...
$ ss local
local        up  2w00d,  4 users, idle 3.00s, load 0.32, %usr 0, %sys 4
```

Finally, *coshell* provides access to the *coshell* daemon. *coshell +* starts the per-user *coshell* daemon if one is not already running. *coshell -* enters an interactive *coshell* daemon query interpreter:

```
$ coshell -
```

```
coshell> t
```

SHELLS	USERS	JOBS	CMDS	UP	REAL	USER	SYS	CPU	LOAD	RATING
6/7	1/9	0/38	60	1d17h	2m36s	4m41s	1m36s	9/9+0	1.75	18.88

```
coshell> s
```

CON	JOBS	TOTAL	USER	SYS	IDLE	CPU	LOAD	RATING	BIAS	TYPE	HOST
8	0	6	55.18s	11.89s	7h06m	1	0.00	21.00	1.00	sun4	kiwi
7	0	12	1m16s	16.38s	2.00s	3	1.20	18.00	1.00	sun4	gryphon
13	0	8	1m03s	17.79s	%	1	1.36	21.00	1.00	sun4	parker
17	0	8	1m04s	13.97s	6h13m	1	1.44	21.00	1.00	sun4	condor
9	0	4	22.95s	36.24s	%	2	4.48	22.00	1.00	sun4	toucan
6	0	0	0	0	1.00s	1	0.40	9.00	4.00	sun4	dodo

```
coshell> j
```

JOB	USR	RID	PID	TIME	HOST	LABEL
6	15	1	16577	7.00s	gryphon	make main.o
7	15	2	18940	7.00s	toucan	make job.o
8	15	3	8348	7.00s	condor	make schedule.o
9	15	4	1385	7.00s	kiwi	make shell.o
10	15	5	12952	7.00s	parker	make command.o
11	15	6	16578	7.00s	gryphon	make misc.o
12	15	7	18942	7.00s	toucan	make init.o

alias on="coshell -r" provides a command level equivalent to *rsh*:

```
# execute hostname on ``best`` host
$ on - hostname
toucan
# execute hostname on the ``best`` host of a different type
$ on '(!type@local)' hostname
lynx
```

6. Performance

Figure 3 shows empirical timings of *nmake* building itself using various combinations of local and network *coshells* and concurrency levels. The *nmake* build compiles and links 18 C files into a single executable.

The network configuration includes many idle sparc 2's rated at 21 mips, more than double the local sparc 1's rating of 10 mips. Compensating for the rating differences, the network *coshell* provided a factor of 4 improvement in compile time. Empirical timings for other projects ranging in size from 10K lines of source (like *nmake*) to 1M lines of source show an average 5 times build time improvement. In particular, the 1M line project build time went from 10 hours down to 2 hours. This factor of 5 has also been noted in other network build implementations.

A detailed study of the limitations on performance has not been done yet, but NFS/RFS network file traffic saturation most certainly plays a role.

```

sun sparc 1, 10 mips, 1 cpu, concurrency 1, local coshell
real      9m30.28s
user      5m44.31s
sys       1m26.10s

solbourne sparc 2, 18 mips, 3 cpus, concurrency 1, local coshell
real      5m27.63s
user      3m02.70s
sys       0m34.75s

solbourne sparc 2, 18 mips, 3 cpus, concurrency 3, local coshell
real      2m26.56s
user      3m16.66s
sys       0m38.41s

sun sparc 1, 10 mips, 1 cpu, concurrency 10, network coshell, 8 hosts
real      1m33.35s
user      0m04.80s
sys       0m04.10s

```

Figure 3. Empirical *coshell* timings

coshell has been operational since mid 1990 and statistics for the author's personal usage have been maintained since November 1990. These are listed in figure 4.

```

number of coshell daemons      134
number of active shell connections 3426
number of user connections to daemon 22459
number of executed jobs      199010
total daemon up time          1Y01M
total real time while jobs executing 4w01d
total job user time           4d19h
total job sys time            3d13h

```

Figure 4. Author's *coshell* usage from 11/90 to 4/93

7. Conclusion

The network *coshell* has consistently provided factors of 5 improvement in build times for *nmake* users. Its simple, non-privileged administration makes it easy to maintain and port and allows any user to take advantage of idle network cycles.

8. Acknowledgements

coshell has been in development since early 1990 and my colleagues in the Advanced Software Technology Department have suffered through its evolution. Included were innumerable kernel crashes caused by buggy socket implementations (these have all been compensated for) and bad job scheduling parameters that brought the sparcs to their knees.* Also of great help were the handful of beta users that put *coshell* to work in the real world.

* I haven't seen a lynch party since 1991.

References

- [AC92] *NetMake*, Aggregate Computing, Inc., 1992.
- [Bers85] Brian Bershad, *Load Balancing with Maitre d'*, Computer Systems Research Group, UC Berkeley, December 1985.
- [BK89] Morris Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.
- [BLL91] Allan Bricker, Michael Litzgow, Miron Livny, *Condor Technical Summary*, University of Wisconsin Computer Sciences Department Technical Report #1069, October 1991.
- [Bour78] S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.
- [DO91] Fred Douglass and John Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software - Practice and Experience, Vol. 21 No. 8, pp. 757-785, August 1991.
- [Feld79] S. I. Feldman, *Make - A Program for Maintaining Computer Programs*, Software - Practice and Experience, Vol. 9 No. 4, pp. 256-265, April 1979.
- [Fowl85] Glenn S. Fowler, *The Fourth Generation Make*, Proc. of Summer 1985 USENIX Conf., Portland, 1985.
- [Fowl90] Glenn S. Fowler, *A Case for make*, Software - Practice and Experience, Vol. 20 No. S1, pp. 30-46, June 1990.
- [HP90] *Task Broker for Networked Environments Based on the UNIX Operating System*, technical data booklet, Hewlett Packard, 1990.
- [Hume87] Andrew G. Hume, *Mk: a successor to make*, Proc. of Summer 1987 USENIX Conf., Phoenix, 1987.
- [KK90] David G. Korn and Eduardo Krell, *A New Dimension for the Unix File System*, Software - Practice and Experience, Vol. 20 No. S1, pp. 19-33, June 1990.
- [LFJ83] *A 4.2BSD Interprocess Communication Primer*, Computer Systems Research Group, UC Berkeley, 1983.
- [Litz87] Michael J. Litzkow, *Remote UNIX: Turning Idle Workstations into Cycle Servers*, Proc. of Summer 1987 USENIX Conf., Phoenix, 1987.
- [Ritc84] D. M. Ritchie, *A Stream Input-Output System*, AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, October 1984.
- [SM89] R. M. Stallman and R. McGrath, *GNU Make - A Program for Directing Recompilation*, Edition 0.1 Beta, March 1989.
- [Stev90] *UNIX Network Programming*, W. Richard Stevens, Prentice Hall, 1990.
- [SUN87] *SunPro: The Sun Programming Environment*, Sun Technical Report, *make: Keeping Files Up-to-Date*, ch. 5, pp 67-83, Sun Microsystems, 1987.
- [SV84] *Augmented Version of Make*, UNIX System V - Release 2.0 Support Tools Guide, pp. 3.1-3.19, April 1984.

Glenn Fowler is a distinguished member of technical staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of *nmake*, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in electrical engineering, all from Virginia Tech, Blacksburg Virginia.

A User-Level Replicated File System

Glenn Fowler, Yennun Huang, David Korn, Herman Rao

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

This paper describes a replication mechanism called *nDFS/REPL*, which is built inside the Multiple Dimension File System (*nDFS*). Layered on top of physical file systems, *nDFS/REPL* substantially increases the reliability and availability of physical file systems by replicating files located on one physical file system onto another physical file system at run time. Implemented at the user level, the system supports *Kernel Independence* and *Application Transparency* in that the semantics and syntax of system calls are preserved; thus, no kernel modification and application re-compilation are needed. To avoid replicating temporary files, *nDFS/REPL* introduces a new notion—a *replicated tree*—by which users are able to define one or more subtrees of a file system that need to be replicated. Replicated trees are maintained by a user-level, per-process name space. *nDFS/REPL* accesses underlying physical file systems using UNIX system calls, rather than the actual access protocols. Hence, it is easier to port the system to different platforms. Finally, in conjunction with a process daemon, the system is resilient to any single-point of failure, either software or hardware. Applications are able to continue processing even when the backup physical file system is unreachable. When the backup file system is available again, a recovery mechanism synchronizes it with the primary file system without terminating running applications. Preliminary testing results show approximately 10% overhead for replication on typical Unix tools.

1 Introduction

The Multiple Dimension File System (*nDFS*) is a virtual logical file system currently being developed by the Software Engineering Research Department in AT&T Bell Laboratories. In addition to the abstractions of file and directory supported by traditional UNIX-like¹ file systems, *nDFS* introduces four file system concepts to UNIX file systems: viewpathing, file versioning, event notification, and tree replication. First, viewpathing allows a logical directory to refer to a sequence of physical directories, the virtual content being an ordered union of files in the physical directories [Korn90]. Second, the mechanism for handling multiple versions of files is embedded in the logical file system itself [Korn90]. Third, the logical file system is able to watch for file access events and notify a remote server, which can trigger actions specified by users [Rose91]. Finally, files under *replicated trees* can be replicated to a *backup* physical file system whenever their context/attributes are changed. With these new concepts, *nDFS* enhances the file systems from two dimensions—a hierarchical tree of files and directories—to multiple dimensions. In fact, the infrastructure of *nDFS* is so general that it is easy to create and implement new dimensions to file systems. This paper details our experience in designing and implementing the replication dimension of *nDFS*, which is called *nDFS/REPL*.

nDFS/REPL increases the availability and reliability of physical file systems by replicating files located on one physical file system—the primary file system—onto another physical file system—the backup file

¹UNIX is a registered trademark of UNIX System Laboratories, Inc.

system. Basically, *nDFS/REPL* is designed for a dual-system architecture, where applications are running on the same host as the primary file system. The ultimate goal of this replication is to maintain the state of the backup file system immediately after the primary file system crashes to be the same as that of the primary file system after it reboots from the crash. Hence, when the primary file system fails, applications can quickly recover by switching over to the host of the backup file system rather than waiting for recovery of the primary file system. In a more general configuration, where applications may be running on a third machine, a layer between the operating system and applications is needed to help applications re-direct existing file access to the backup file system when the primary file system fails.

Conceptually, *nDFS/REPL* introduces a logical layer between the operating system and applications. This layer intercepts system calls from applications that modify files/directories (e.g., `creat`, `write`, `chmod`, etc.) and propagates these calls to a remote server, which in turn performs them on a backup physical file system.

Many Distributed File Systems, such as Coda [Saty90], Deceit [Sieg90], Echo [Hisg89], HA-NFS [Bhid91], and Ficus [Guy90], implement replication mechanisms on file servers embedded in the operating system. All these file systems are homogeneous—they define and use their own access protocols. *nDFS/REPL* is unique in that it is layered on top of physical file systems and implemented on the client side rather than on the file server. *nDFS/REPL* requires no special hardware, such as Mirrored Disk [Bitt88]. By moving functionality of replication to the end-point, i.e., the file system user, *nDFS/REPL* supports:

- **Kernel Independence and Application Transparency**

The system is designed and implemented at the user level in such a way that the operating system and applications need not be changed. That is, no kernel modification is needed. The syntax and semantics of system calls are preserved². To intercept system calls, *nDFS/REPL* replaces the standard C library (i.e., `libc`) with a new library. For systems providing the concept of dynamic shared libraries, e.g., Sun OS 4.1 [Sun88], applications may invoke the new library simply by defining the library search path. For systems without dynamic shared libraries, applications must be re-linked with the new library.

- **Fine-Grain Replication**

To improve performance, *nDFS/REPL* allows users to specify those files that need to be replicated through replicated trees. Replicated trees are maintained by a user-level name space provided by *nDFS/REPL*. Only files under a replicated tree are replicated. By doing so, *nDFS/REPL* reduces the costs of replicating the entire file system, something many other replicated distributed file systems do, e.g., Deceit [Sieg90] and HA-NFS [Bhid91].

- **Portability**

nDFS/REPL is a logical file system layered on top of existing physical file systems. It uses standard UNIX system calls to access underlying physical file systems, regardless of what access protocols are provided by file servers. In addition, we take advantage of highly portable libraries, such as `libsfio` [Korn91]. Hence, it is easy to port the system to different platforms. Furthermore, *nDFS/REPL* is capable of running on heterogeneous file system environments. For example, the primary file system might be a regular UNIX file system, while the backup file system might be running Sun Network File System [Sand85].

- **Robustness**

In conjunction with a process watch mechanism called Watchd Daemon [Huan93], the replication mechanism is capable of recovering from a single failure. Failure and recovery are transparent to applications in that a failure does not force operations in progress to terminate. An on-line recovery mechanism synchronizes the backup file system with the primary file system without interrupting running applications.

We have implemented a prototype of *nDFS/REPL*, running on Sun OS 4.1, HP-UX, SVR4, and SGI MIPS. Several AT&T projects use *nDFS/REPL* in a dual system-architecture to decrease system down time. Much effort has been spent to improve performance. The majority of the overhead of this logical file system comes

²We do, however, embed operations to interact with the logical file system in the system call mount.

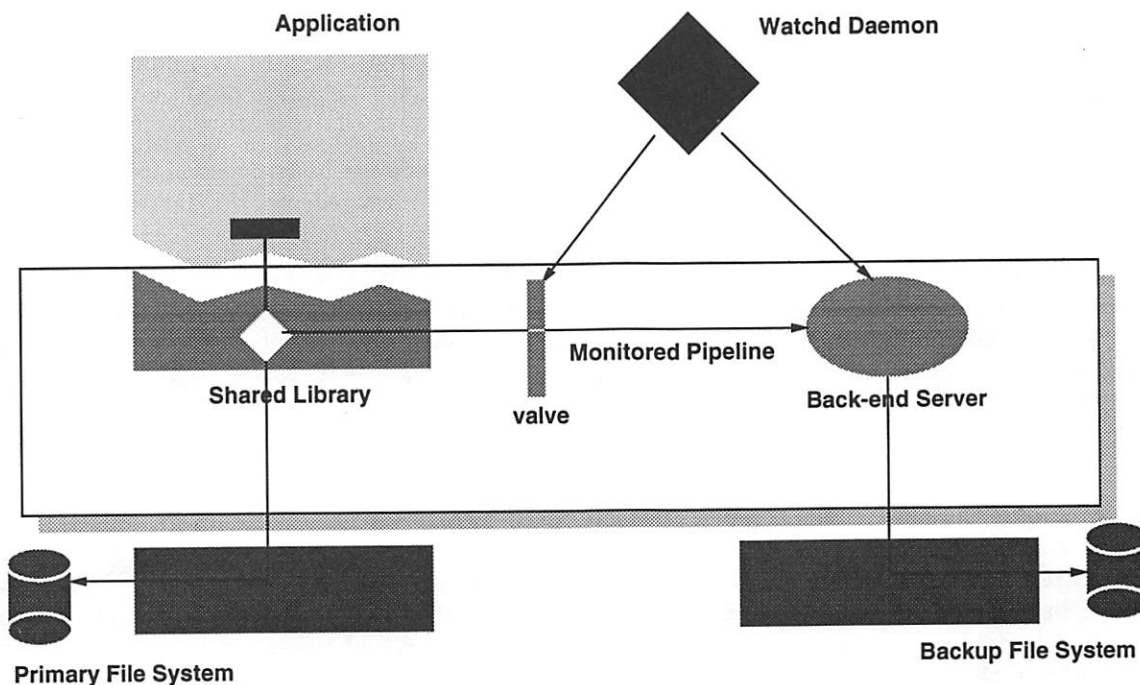


Figure 1: System Architecture

from the replication mechanism, which is proportional to the number of write calls invoked by applications. Performance evaluations show the following three points:

- For applications with *Intensive-Write* operations (e.g., *cpio*, and *pax*), the overhead is less than 25%;
- For applications with *Moderate-Write* operations (e.g., *make* and *cc*), the overhead is approximately 10%;
- For applications with *light-write* operations (e.g., *vi*), the overhead is insignificant—approximately 1%.

The paper is organized as follows: Section 2 presents the system architecture. Section 3 overviews the implementation. Section 4 describes how *nDFS/REPL* recovers from a single failure. Section 5 offers discussions. Finally, Section 6 concludes the paper.

2 System Architecture

Conceptually, *nDFS/REPL* contains four major modules: a shared library that intercepts system calls invoked by an application, a back-end server that replays those system calls on a backup file system, a monitored pipeline that supports a reliable communication channel between the shared library and the back-end server, and a Watchd Daemon that monitors the system processes. Figure 1 illustrates the system architecture.

We have used two fault tolerance techniques: Watchd Daemon and libft [Huan93]. Watchd is a daemon that detects failures of registered applications by polling them or catching their exit signals from operating systems³. When a failure is detected, Watchd restarts the failed application by default, or executes actions

³To catch the exit signals, applications are in fact started by Watchd.

specified by the application. The reliability of Watchd itself is achieved by implementing it as three mutual-watching processes, each of which is capable of detecting and recovering failure of other two processes. Furthermore, a Watchd located on a host is able to watch another Watchd located on a different host. The result is that a sequence of Watchds located on multiple hosts may form a chain to detect node failures in a distributed environment.

libft is a library that provides programmers with two basic functions to recover failure: `checkpoint()` which copies the critical data onto external storage, and `recover()` which recovers the data to the previous checkpoint.

The shared library presents applications with a nearly complete UNIX system call interface. It passes system calls invoked by applications to the operating system. In addition, it also selects the system calls that modify files under a replicated tree and sends them down a monitored pipeline. In a normal situation, the other end of the pipeline is the back-end server, which then replays these system calls on a backup file system. The back-end server may be located either on the same host as the applications, or on a different host. The communication between the shared library and the back-end server is asynchronous for better performance.

From an application's point of view, a write system call invokes a write operation to the primary file system and another write operation to the backup file system. The former is a write-through operation, while the latter is in fact a write-behind operation. To synchronize a file on the backup file system with the one on the primary file system, *nDFS/REPL* modifies the system call `fsync`. In addition to its normal functionality, the `fsync` call requests an acknowledgment from the back-end server and blocks until receiving it. A return from the `fsync` guarantees that the named file in the backup file system has the same state as the one in the primary file system.

The pipeline between the library and the back-end server is monitored and controlled by the Watchd Daemon, which is running as a third process. When the Watchd Daemon detects that the back-end server is unreachable, because of the failure of either the back-end host or the communication channel, the system turns the *valve* of the pipeline to a different direction, as illustrated in Figure 2. The result is that input to the pipeline, system call messages from the shared library, are redirected and logged into a file located on the primary file system.

When the back end is available again, the Watchd Daemon turns the valve of the pipeline back to the normal position, as illustrated in Figure 1. Before the back-end server functions normally, it synchronizes the backup file system by first replaying system calls stored in the log file located on the primary file system.

The shared library employs a *retry* protocol to deal with the change of the valve. That is, when the valve is turned to the other direction, the shared library becomes aware of changing the valve because of a write failure. By simply re-consulting the server handling the pipeline (described in the next section), the shared library gets a new handle of the pipeline, i.e., a file descriptor, and writes the data. However, the shared library has no knowledge of what is on the other end of the pipeline. It is important to note that the process of turning the valve is transparent to applications in that it does not terminate running processes.

Single Pipeline

All applications on a host share a single pipeline to the back-end server, rather than separate ones. There are two reasons for this design decision. First, by sharing a per-host pipeline, applications avoid the cost of connecting to the back-end server. The pipeline is created and maintained by a local server, which issues the file descriptor of the pipeline to each application. Second, a per-host pipeline is easier to maintain, in terms of failure and recovery, than multiple per-application or per-process pipelines.

User Level Name Space

nDFS/REPL allows users to define one or more subtrees of the file system as replicated trees. Files under a replicated tree are replicated whenever their context/attributes are changed. A trace analysis shows that

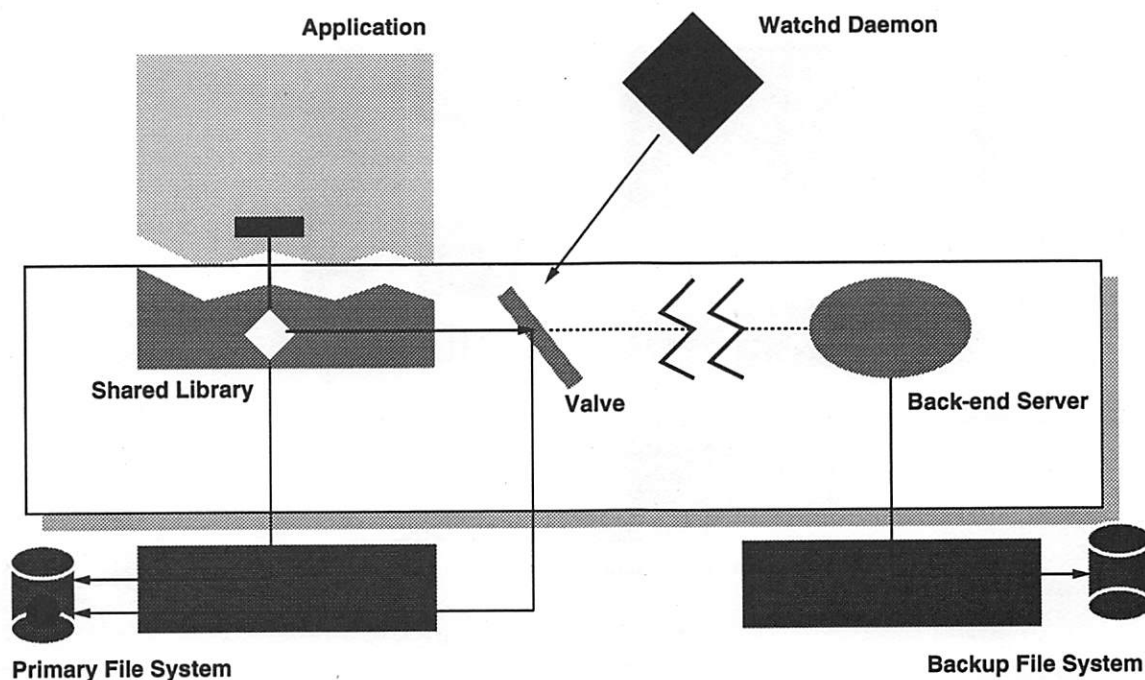


Figure 2: Failure

50-60 percent of files in UNIX file systems are temporary working files with a lifetime of less than 3 minutes [Oust85]. Replicated trees allow users to avoid replicating these temporary files, and therefore improve overall performance.

The notion of replicated trees is implemented by a user-level name space supported by *nDFS/REPL*. By user-level, we mean that the name space resides on the user address space of a process and is inherited by its child processes at the fork and exec operations. The name space maintains a prefix table [Welc86] to map a pathname to another pathname. A tree is a replicated tree if its corresponding pathname in the prefix table refers to a back-end server.

3 Implementation

We have implemented a prototype of *nDFS/REPL* on Sun OS, SVR4, HP-UX, and SGI MIPS. This section first overviews the system components and then examines the performance of the prototype.

System Components

As illustrated in Figure 3, *nDFS/REPL* consists of six major software components:

- Library (lib3d)⁴: implements the shared library;
- Pipeline Server (PipeSvc): maintains the monitored pipeline;
- Back-End Logging Server (BackLog): logs system call messages from the pipeline;
- Syscall Engine (SysEng): replays system calls from log files;

⁴The library is called lib3d instead of libnd for historical reasons.

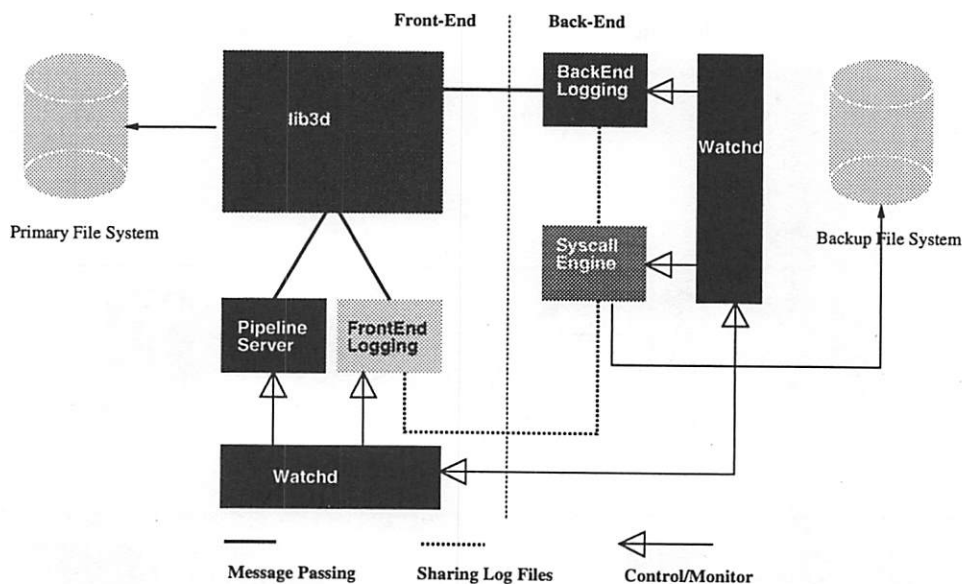


Figure 3: System Structure

Front-End Logging Server (FrontLog): logs messages to the primary file system;

Watchd Daemon (Watchd): detects failures and restarts processes.

In Figure 3, solid lines refers to message passing between components; dotted lines represent sharing of log files among two components; and arrowed lines point to the components monitored by Watchd.

As a replacement of libc, lib3d presents applications with UNIX system call interface. In addition to passing system calls to the operating system, lib3d selects the system calls that modify files/directories under a replicated tree, and passes them on the pipeline. The protocol between lib3d and the back end server is defined according to UNIX file system semantics rather than any particular file access protocol, such as NFS. The binding between applications and lib3d can be either static, i.e., at linking time, or dynamic, i.e., at run time.

A file descriptor (FD), rather than a pathname, is used to refer to a file in some system calls, such as fstat and write. Unfortunately, FDs have context only within a process. Moreover, UNIX allows FDs to be duplicated and inherited by a child process. Thus, we need some general identifiers for files without complicating lib3d. Our solution is to use the *device* and *inode*, which can be retrieved from the system call stat or fstat, to refer to the desired file. The back-end in fact maintains a table mapping *device/inode* to pathnames.

The challenge in implementing such a library is to preserve the syntax and semantics of system calls, including error return codes (i.e., *errno* in UNIX). On the other hand, the goal of the replication mechanism is to maintain a replicated sub-file system on a backup file server that is identical to the original one in the primary file server. In particular, we need to consider the following:

- Only those calls that are successfully executed in the primary file need to be sent to the back-end.
- In addition to input arguments of system calls, more information is needed. For example, the system call *creat* takes only two arguments: *pathname* and *mode*. But to create a duplicated file on the backup file system, SysEng needs the owner and group ID of the file. More important, the exact *mode* of the newly created file should be sent to the back-end, rather than the input arguments of the system call *creat*.

- To re-produce the same result on the backup file system, system calls other than the one invoked by the application may be needed. For example, an application renames a file located outside the replicated tree to the one under the tree with the system call `rename`. In this case, `lib3d` must pass to the back-end server the system call `creat` and a sequence of the system call `write` to re-create the file on the backup file system.

The system is designed with the assumption that the trees to be replicated on the primary file system and on the backup file system are identical (i.e., with same directories and files) at the initial state.

The back-end server described in the previous section is implemented as two separate processes: `BackLog` and `SysEng`. The former receives messages about system calls from the monitored pipeline and writes them into log files. The latter then reads in messages from those log files and executes the corresponding system calls. The reason for splitting the back-end server into two processes is to make `BackLog` as simple as possible. By moving the function of executing messages to another process, `BackLog` performs only a very simple task: copy incoming messages from the pipeline to a log file. It runs fast and avoids making the pipeline the bottleneck of the system. Also, it is robust and thus makes the pipeline more reliable. Moreover, `BackLog` and `SysEng` may share more than one log file. As a result, it is easy to do garbage collection, and failure recovery is fast.

`PipeSvc` maintains the monitored pipeline between the front-end and the back-end. Initially, it creates a connection to `BackLog` using TCP/IP or Streams through the connect stream library (`libcs`). We take advantage of UNIX Domain socket that allows passing file descriptors from one process to another. `PipeSvc` functions as following:

- Whenever `lib3d` requests the pipeline, `PipeSvc` returns the file descriptor of the connection.
- When `Watchd` detects back-end failure, `PipeSvc` turns the valve of the pipeline by closing the connection to `BackLog` and opening a new connection to `FrontLog`.
- When `Watchd` detects back-end recovery, `PipeSvc` turns back the valve by closing the connection to `FrontLog` and opening a connection to `BackLog`.

To make the connection more reliable, `PipeSvc` also deposits the file descriptor of the connection with `Watchd`. When `PipeSvc` itself fails, a restarted `PipeSvc` can then retrieve the current connection from `Watchd`. The process of turning the valve back and forth is not completely hidden from `lib3d`, which is aware of the change when fail of writing to the pipeline. It then gets the new connection from `PipeSvc`. The communication between `lib3d` and `BackLog` is asynchronous for better performance.

Similar to `BackLog`, `FrontLog` records messages from `PipeSvc` to the primary file system. It is started by `Watchd` when the back-end fails, and it is terminated when the back-end recovers. The log file is used for recovering the backup file system.

Finally, `nDFS/REPL` relies on `Watchd` to detect failure of system components. `Watchd` is a per-host based service, and `Watchd` on the front-end and the back-end also watch each other. Section 4 discusses failure and recovery in more detail.

Performance Evaluation

To understand the overhead of the logical layer, we have measured the performance of the prototype using a variety of applications. The hardware for testing includes a Sun 4/75 named `condor`, and a Solbourne 5/800 named `gryphon`, connected by a 10 Mbps Ethernet. Both machines are running Sun OS 4.1.1.

We have evaluated three cases, each of which consists of applications running on `condor`. In the first case, applications access files located on the local disk, i.e., on `condor`, with the standard library `libc`. In the second case, applications access files located on the remote file server, i.e., on `gryphon`, again with the standard library `libc`; the Sun Network File System is used to access remote files. In the third case, applications run

on *nDFS/REPL*: the primary file system is located on the local disk and the backup file system is located on gryphon.

The majority of the overhead comes from the replication mechanism, which is proportional to the number of write calls invoked by the application. According to the number and size of write calls invoked, we have classified applications into three categories: *Light-Write Applications*, *Moderate-Write Applications*, and *Intensive-Write Applications*. The benchmark for testing includes three phases for different categories: in the Light-Write phase, we concatenate files using *cat*, scan every byte using *wc*, and output the result to a file; in the Moderate-Write phase, we compile a big source file using *cc*. and in the Intensive-Write phase, we unravel an archive file using *cpio*.

	UNIX FS	NFS	<i>nDFS/REPL</i>
Light-Write	7.75 sec	7.80 sec	7.78 sec
Moderate-Write	13.01 sec	15.00 sec	14.08 sec
Intensive-Write	6.31 sec	12.70 sec	7.72 sec

Table 1: Performance Results

As Table 1 shows, in the Light-Write category, the difference among three cases is insignificant. In the Moderate-Write category, *nDFS/REPL* exhibits an 8% slow down compared to the UNIX File System case. But it has better performance than the NFS case. This is because *nDFS/REPL* accesses files located on the local disk and uses asynchronous communication to pass data to the backup file system. The NFS case, on the other hand, accesses file on the remote file server and synchronously writes data back to the remote file server. Finally, in the Intensive-Write category, the replication in *nDFS/REPL* costs 23% overhead compared to the UNIX File System case. However, *nDFS/REPL* is 40% faster than NFS.

To further understand throughput for write system calls, we have examined these three cases with two programs that measures writing of 1 Kbytes data and of 100 Kbytes data 100 times repeatedly, as listed in Table 2. In the first part, the *nDFS/REPL* throughput is 68% of that of the UNIX File System case. This is because for each write call, *nDFS/REPL* needs to do an extra write to the pipeline. When the size of data becomes as big as 100K in the second part, *nDFS/REPL* needs to fragment data and send out multiple writes to the pipeline, because the buffer size of the pipeline is limited. The throughput is 49% of that of UNIX case. *nDFS/REPL* has better throughput than NFS in both cases. Again, this is because of the synchronous write used by NFS.

	UNIX FS			NFS			<i>nDFS/REPL</i>		
	sec	Kb/s	%	sec	Kb/s	%	sec	Kb/s	%
1K x 100	0.21	476	100	0.50	200	42	0.31	322	68
100K x 100	2.12	472	100	9.85	102	22	4.32	231	49

Table 2: Throughput

These results lead to two conclusions:

- Compared to the UNIX File System, the overhead due to replication in *nDFS/REPL* is around 10% in typical UNIX applications and less than 25% for for the Intensive-Write tools.
- The performance of accessing files on a UNIX File System while replicating files onto a remote file system is better than that of directly accessing files on the remote file server using NFS.

4 Failure and Recovery

This section examines how *nDFS/REPL* tolerates failures, and explains the strategy used to recover from those failures. This design has two chief objectives:

- To ensure that the overhead of recovery, in terms of performance, is minor.
- To ensure failure and recovery are transparent to applications and that running applications are not terminated.

The recovery procedure is based on the assumption that *Watchd* is the most reliable component in the system. This is because it performs a very simple task and is capable of self-recovery after failure. Consider the following failure scenarios:

- **PipeSvc fails:**
Watchd detects the failure and restarts the server. The newly restarted *PipeSvc* retrieves the connection from *Watchd*. No other processes are aware of this failure/recovery.
- **SysEng fails:**
Watchd detects the failure and restarts *SysEng*. With checkpoint and recovery functions provided by *libft*, the newly restarted *SysEng* is able to recover to its previous-checkpoint status from an external file. No other processes are aware of this failure/recovery.
- **BackLog fails:**
Watchd detects the failure and restarts the *BackLog*. Again, *BackLog* restores its status from a checkpoint file. *Watchd* also informs *PipeSvc* of the change, which then updates the monitored pipeline to connect to the new *BackLog*. The next write of each application fails and *lib3d* gets the new connection from *PipeSvc*.
- **Back-end fails:**
Watchd on the front-end detects the failure. *Watchd* informs the *PipeSvc*, which then activates *FrontLog* and turns the valve of the monitored pipeline to connect to *FrontLog*. The next write of each application fails and *lib3d* gets the new connection from *PipeSvc*.
- **Back-end system is recovered:**
Watchd located on the back-end restarts *BackLog* and *SysEng* and informs its peer *Watchd* on the front-end system of its status. The latter then terminates *FrontLog* and informs *PipeSvc* to turn back the valve of the pipeline by reconnecting to *BackLog*. *SysEng* copies and executes the log files created by *FrontLog* during the down time. After that, *SysEng* is ready for normal operations.
- **FrontLog fails:**
Watchd informs *PipeSvc*, which then restarts *FrontLog* and reconnects to the new *FrontLog*. Notice that *FrontLog* exists only when the back-end fails. The next write of each application fails and *lib3d* gets the new connection from *PipeSvc*.

TCP/IP is used for communication between the front-end and the back-end. Message-passing between *lib3d* in the front-end and *BackLog* in the back-end is asynchronous for better performance. However, to detect that a message has been lost because of instant failure of the back-end, *lib3d* sends a dummy message right after the regular message. This is because the success of sending a message does not guarantee the delivery of the message on the other end; the success of sending the *second* message does, however, guarantee the delivery of the *first* one. If the regular message is not delivered, a failure procedure, as described above, takes place. A simple experiment has shown the performance ratios of asynchronous send-once, asynchronous send-twice, and synchronous send-reply to be 1, 1.4, and 2.1.

5 Discussion

End-Point Replication

Compared to other distributed file systems, such as Coda [Saty90], Deceit [Sieg90], Echo [Hisg89], HA-NFS [Bhid91], and Ficus [Guy90], *nDFS/REPL* is unique in that it is layered on top of the operating systems and physical file systems and is implemented on the client side without any modification on file servers. It is general enough that it can be run on top of a variety of physical file systems. Portability, heterogeneity, and Kernel Independence and Application Transparency are the major advantages of this *Client-Central* approach.

Monitored Pipeline

The notion of a pipeline is a common abstraction for Inter-Process-Communication (IPC). We have enhanced pipeline's functionality by adding a *valve* to a pipeline. A monitored pipeline contains two receivers, and the valve is used to determine which receiver gets the input from the pipeline. In *nDFS/REPL*, a failure of the receiver turns the valve of the pipeline to a different direction. In addition, a retry protocol is adopted by the sender to switch from one receiver to the other. As with regular pipelines, the sender of a monitored pipeline has no knowledge about the other end of the pipeline. Thus, monitored pipelines are capable of handling failure and recovery.

Consistency

Conceptually, a write function call in an application is translated into a write-through operation to one file system (i.e., the primary file system) and a write-behind operation to another file system (i.e., the backup file system). Because of the write-behind operation, *nDFS/REPL* guarantees sequential write sharing [Nels88] only when all applications are using the same file system as the primary. Currently, we are designing and implementing a token server, which maintains per-file tokens between two file systems. A given file on the file system **A** is accessible only if **A** owns the access token and there are no outstanding write-behind operations to this file on **A**. Note that even though there are outstanding write-behind operations to the file on the file system **B**, the file on **A** may be still accessible. Providing maximum concurrency with acceptable performance is the major challenge in designing such a token server.

6 Conclusion

This paper introduces a simple and practical mechanism for replicating files located on one physical file system into another physical file system. The solution is practical in that it requires no modification of operation systems and application programs and in that it needs no special hardware, such as Mirrored Disk. The robustness of the system is achieved by well-defined failure semantics. The system has been ported to a variety of platforms and has been adopted by several projects in AT&T.

Acknowledgments

Chandra Kintala made valuable comments on earlier drafts of this paper and the work it describes. Rao Arimilli, Randy Cramp, and Robin Knight provided helpful discussions and feedback from testing the prototype.

References

- [Bhid91] Bhide, A., Elnozahy, E., and Morgan, S. A highly available network file server. In *Usenix Conference Proceedings*, pages 199–205, January 1991.
- [Bitt88] Bitton, D. and Gray, J. Disk shadowing. In *Proceedings of 14th Conference on Very Large Data Bases*, pages 331–338, September 1988.
- [Guy90] Guy, R., Heidemann, J., Mak, W., Page, T., Popek, G., and Rothmeier, D. Implementation of the Ficus replicated file system. In *Usenix Conference Proceedings*, pages 63–71, June 1990.
- [Hisg89] Hisgen, A., Birrell, A., Mann, T., Schroeder, M., and Swart, G. Availability and consistency tradeoff in the ECHO distributed file system. In *Second Workshop on Workstation Operating Systems*, pages 49–54, September 1989.
- [Huan93] Huang, Y. and Kintala, C. Software implemented fault tolerance. In *1993 Fault Tolerant Computing Symposium (FTCS23)*, June 1993.
- [Korn90] Korn, D. and Krell, E. A new dimension for the Unix file system. *Software—Practice and Experience*, 20(S1):S1/19 – S1/34, July 1990.
- [Korn91] Korn, D. and Vo, K.-P. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer Usenix*, 1991.
- [Nels88] Nelson, M. N., Welch, B. B., and Ousterhout, J. K. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [Oust85] Ousterhout, J. K., Costa, H. D., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A Trace-Driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 15–24, December 1985.
- [Rose91] Rosenblum, D. S. and Krishnamurthy, B. An event-based model of software configuration management. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 94–97. ACM SIGSOFT, 1991.
- [Sand85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun Network File System. In *Proceedings of Summer Usenix*, pages 119–130, June 1985.
- [Saty90] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 39(4):447–459, April 1990.
- [Sieg90] Siegel, A., Birman, K., and Marzullo, K. Deceit: A flexible distributed file system. In *Proceedings of Summer Usenix*, July 1990.
- [Sun88] Sun Microsystems, Inc., Mountain view, Calif. *Shared Libraries*, May 1988.
- [Welc86] Welch, B. B. and Ousterhout, J. K. Prefix tables: a simple mechanism for locating files in a distributed system. In *Proceedings of the 6th Conference on Distributed Computing Systems*, pages 184–189, May 1986.

Glenn Fowler is a distinguished member of technical staff in the Software Engineering Research Department with AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of **nmake**, a configurable ANSI C preprocessor library, and the **coshell** network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in electrical engineering, all from Virginia Tech, Blacksburg Virginia.

Yennun Huang received his B.S. in electrical engineering from National Taiwan University in 1982, and M.S. and Ph.D. in Computer Science from University of Maryland at College Park in 1986 and 1989,

respectively. He has been with the AT&T Bell Labs research since 1989. Currently, he is a member of technical staff in AT&T Bell Labs at Murray Hill. His research interests are fault tolerant computing, software engineering, distributed systems and performance evaluation.

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer). He is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, ksh, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs fellow award.

Herman Chung-Hwa Rao is a member of technical staff at Murray Hill, Bell Laboratories. His research interests are in the area of Distributed Computing Environments, Distributed File Systems, and Distributed Operating Systems. In particular, he has been invoked in the design and implementation of Jade File System, a User-Level Replicated File System (nDFS/REPL), and a Distributed Application Framework (DAF). He received his Ph.D. and M.S. in Computer Science from the University of Arizona and his B.S. in Mechanical Engineering from the National Taiwan University.

sfs: A Parallel File System for the CM-5*

Susan J. LoVerso
William Nesheim

Marshall Isman
Ewan D. Milne

Andy Nanopoulos
Richard Wheeler

CM-5 Operating System Group
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1264

Abstract

This paper describes the creation of a UNIX-compatible file system with highly scalable performance and size. The file system is on the CM-5 backed by a scalable array of disks. Using the UNIX file system (UFS) from the SunOS 4.1.2 kernel as a base and modifying it to support Connection Machine (CM) operations, we have created a new file system, the scalable file system, or *sfs*. We discuss the CM operations we support, such as parallel reads and writes to the processing nodes of the Connection Machine, the use of NFS to support many partitions of processing nodes on the CM, support for very large file sizes (64-bit) and support for odd numbers of disk drives. The tradeoffs and decisions made during the course of this project as well as performance data for varying numbers of disk drives are provided.

1 Introduction

The data parallel programming model is the predominant programming paradigm on the CM-5. Typical applications deal with tremendously large data sets. When hundreds or thousands of SPARC processors require access to this data simultaneously, a high bandwidth I/O mechanism is necessary. The CM-5 requires a mass storage system that can support terabytes of data and transfer speeds of hundreds of megabytes per second. With today's technology, no single disk can fulfill those requirements; by using multiple disks and striping the data across all of them, we can meet the data rates required. We have created a highly scalable disk array (SDA) that provides great flexibility in I/O capabilities as well as flexibility in file system organization. These systems with the hardware and software described in this paper are currently running at dozens of customer sites.

CMOST, the operating system that controls the CM-5 derives directly from SunOS 4.1.2. It runs on the control processor (CP), which is a SPARCstation that controls access and use of the Connection Machine processing nodes (PN). The control processor regulates access to I/O devices from the PNs. Since the CP runs a derivative of SunOS, we wanted our file system to have UNIX-compatibility. Using UFS[5, 3] as a starting point, we created our own file system, called *sfs*, which is backed by the SDA. One of the primary functions that *sfs* serves is to provide the CP

*CM-5 is a registered trademark of Thinking Machines Corporation.

UNIX is a registered trademark of Unix System Laboratories.

NFS, SunOS, SPARC and SPARCstation are trademarks of Sun Microsystems, Inc.

with the information necessary to allow parallel I/O from the processing nodes to the SDA. UFS was inadequate for our needs in several ways:

1. It does not have support for file system block sizes and fragment sizes that are not a power of 2.
2. It does not support our parallel calls.
3. It does not have support for files larger than 2 gigabytes.
4. For files that are many megabytes in size, it produces highly fragmented files.

Due to the volume of changes required to support these four items, we chose to create a new file system type instead of making the changes in UFS.

In this paper, Section 2 presents a general hardware overview of the CM-5 and Section 3 gives an overview of the SDA architecture. These give the reader an understanding of how the hardware we are dealing with impacts the decisions we made. Then, Section 4 discusses the basic mechanics of parallel I/O and how it interacts with the rest of the system. Then Section 5 continues with a discussion of the support and design changes made to our *sfs* file system and to NFS in Section 6. Section 7 shows performance data showing the capability of this system. Section 8 concludes with a summary and future work.

2 CM-5 Architecture Overview

2.1 Hardware Architecture

The CM-5 may have tens to thousands of processing nodes. Each node has its own memory and may be executing programs written in either SIMD-style or MIMD-style. Access to the nodes is supervised by the Control Processor.

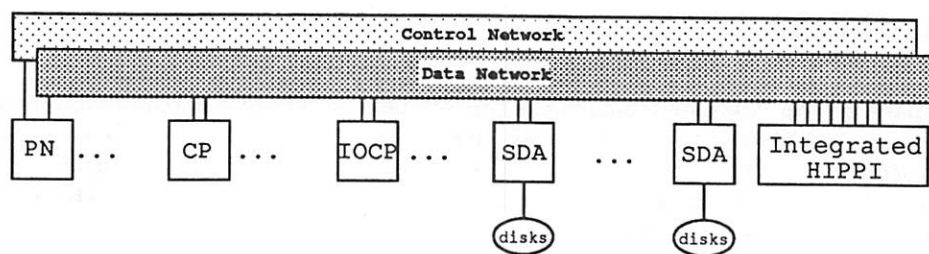


Figure 1: CM-5 Hardware Overview

Two scalable communication networks connect all CPs and PNs as shown in Figure 1. The Control Network handles communications that affect or involve all the nodes, such as broadcasts or synchronization operations. The Data Network handles data traffic – generally point-to-point operations. The SDA is also attached to these networks. The PNs and CPs communicate via the Control Network and all the I/O related activity occurs over the Data Network. The network design provides low latency for transmissions to near neighboring addresses, while preserving a high, predictable bandwidth for more distant communications[10]. The CP on which the SDA file system resides is designated as the IOCP.

High speed external networks, such as HIPPI, are connected directly to the Data Network. Slower external networks, such as Ethernet or FDDI, connect into the CM-5 via the control processor.

In order to provide a scalable high performance disk subsystem, one must spread data across multiple disks. For protection of data against disk failures, the SDA uses a Redundant Array of Inexpensive Disks (RAID) [7, 9] model. The Data Network within the CM-5 carries 20-byte packets for I/O; a 4-byte header precedes 16 bytes of data. A RAID level 3 implementation implies that a small amount of data is written to each disk, and parity data is written for each *disk stripe* of data. With our packet size, it is natural to choose 16 bytes as the amount of data written to each disk before moving on to the next. With each stripe of data, 16 bytes of parity information is stored on an additional parity disk. The parity disk allows reconstruction of any data which is missing due to a disk failure. This RAID-3 implementation is geared toward reads and writes of large amounts of data.

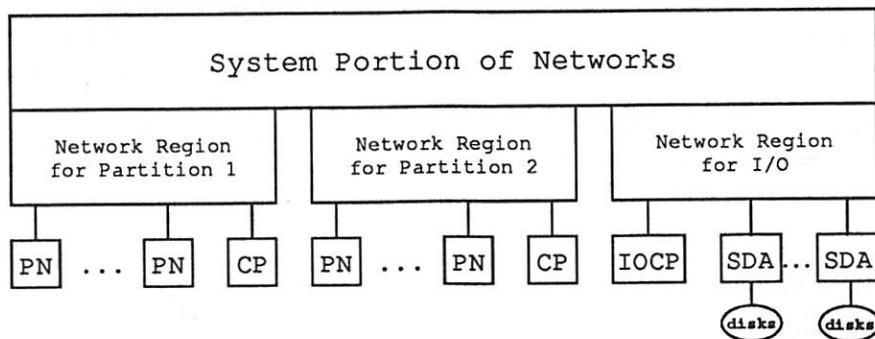


Figure 2: Partitioning the CM-5

The processing nodes of a CM-5 system can be configured into one or more partitions as shown in Figure 2. Each partition is assigned a partition manager – a CP that bears the responsibility for managing the process executing in that partition. The operating system configures the Control Network and Data Network to match the partition structure.

The I/O devices and IOCPs reside in areas of the network address space which are outside any specific partition. Therefore, the I/O devices appear as a global shared resource. There is always additional network capacity for carrying data between partitions and I/O devices. Hence, system-wide data traffic does not interfere with or impede traffic that stays within a partition.

2.2 Software Architecture

A parallel program runs within a partition consisting of a CP and a set of PNs. When performing file system operations, the CP is the client requesting the I/O operation and the IOCP is the file server. The client/server model is well suited to this environment because there can be many partitions sharing the same file system. The control part of the transaction is handled by the NFS protocol. (See Section 6 for more details). The data, however, flows directly between the PNs and the I/O devices over the Data Network. The separation of control and data is done because of the very different latency versus bandwidth tradeoffs required by the two types of data movement. The separation of control and data is similar to the mechanism used on the CM-2 I/O system[2] and the IEEE Mass Storage System Reference Model[1].

A program called the *time sharing daemon* runs on the CP and controls processes requesting

CM-5 resources. It works in conjunction with the UNIX kernel and schedules processes to run on the PNs. In some ways, it can be considered an extension of the kernel. When a CM-5 process requests an I/O operation to the SDA, the time sharing daemon acts on its behalf. The time sharing daemon, using the user's file descriptor, makes a parallel I/O system call to the kernel to access the SDA via the file system. However, the time sharing daemon and the PNs are wholly responsible for the actual I/O operation. This file system transaction will be covered in more detail in Section 4 and Section 5. Now let's take a closer look at the architecture of the SDA.

3 SDA Architecture

One or more units known as Disk Storage Nodes (DSN) comprise the CM-5 disk subsystem. As illustrated in Figure 3, each node contains eight 1.2 gigabyte SCSI disk drives, four SCSI channels, an 8 Mb data buffer, a 20 Mb/second Network Interface and a SPARC processor with its own 8 Mb memory. Each DSN runs a controller microkernel that mediates the I/O between the CM-5 and the drives. DSNs are packaged in units of three as backplanes. Therefore, systems have disks in quantities of 24, 48, 72, etc. Typical configurations might use 22 or 46 or 70 data disks, one parity disk, and one spare disk. By adding more DSNs, you add a balanced amount of bandwidth from the disk through the network interface. By increasing the number of Data Network addresses, which happens automatically when you add more DSNs, you increase the overall Data Network bandwidth as well.

As shown in Figure 1 in Section 2, the SDAs are connected to the Data Network. Each DSN occupies enough network address space to guarantee its sustained 20 Mb/second transfer rate to anywhere in the CM-5.

3.1 Logical Devices

The file system interacts with an abstract virtual disk implemented on top of a set of disks. This set of disks is termed a *logical device*. A Logical Device (LD) consists of an arbitrary number of physical disks grouped together into a RAID 3 disk system. Data is striped 16 bytes per disk. The LD appears as one large disk to the file system. The system administrator edits an ASCII file to select which physical disks become a member of a logical device. There may currently be up to 255 data disks and an optional parity disk in the LD. The configuration of the logical devices is downloaded to the CMOST kernel and to all the DSN kernels running on each of the DSN controllers.

The configuration information describes each logical device. There are four components of the system that require this information:

1. The device driver in the CMOST kernel.
2. The DSN microkernel.
3. The timesharing daemon.
4. The PN microkernel.

The "master" version of the configuration information resides in the CMOST kernel. All other components make queries to the kernel to retrieve it. We use a timestamp mechanism[4] to maintain consistency of this information across the system. An example of some of the information maintained in the configuration tables includes:

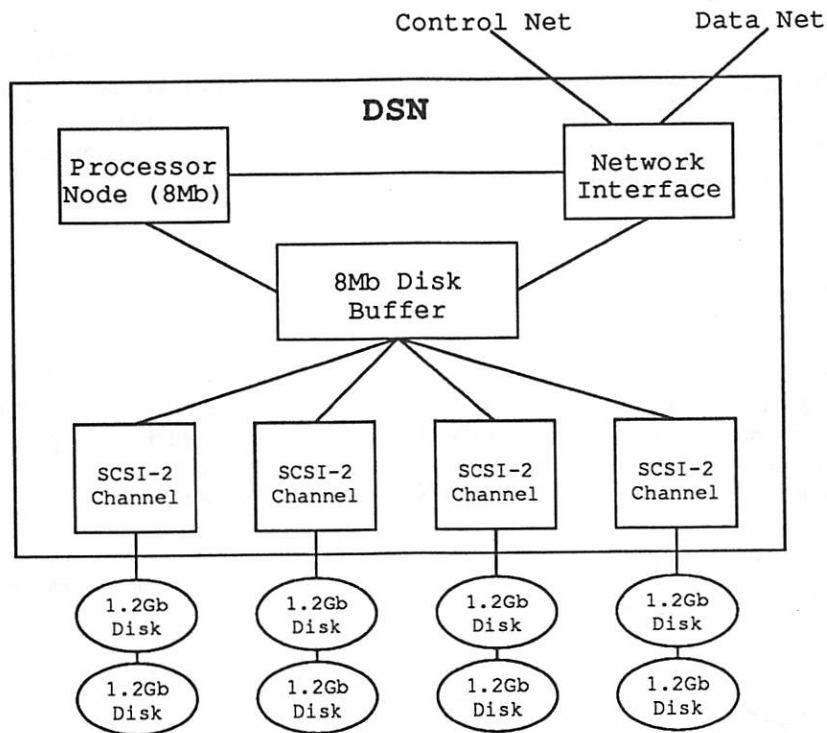


Figure 3: DSN board

- The number of disks in the logical device.
- The network addresses of those disks.
- Whether this logical device has a parity drive.
- The timestamp associated with the logical device.
- A flag indicating the device's state (such as online).
- The network address of the IOCP for this device.
- The `dev_t` representing this device.

Logical devices are global across the CM-5. Lookups of this information can be done using the `dev_t`.

3.2 DSN Microkernel

The SPARC processor on the DSN board runs code which serves as the disk controller firmware for the drives connected to that DSN board. The controller microkernel translates I/O requests for a logical device into specific SCSI controller commands. If a logical device spans more than one DSN, then one DSN is the master of the LD and the other DSNs are slaves. An I/O request is directed to the master DSN which forwards the command to the slaves.

For a read operation, the master and slaves transfer their data asynchronously. As the slaves finish, they send a done message to the master. The master sends a done message to the source

when all DSNs finish. For a write operation, the master and each slave enable their hardware buffers to receive the data. Each slave sends a message to the master indicating how much data it can receive. The master sums this information and sends the total to the source. The source then transmits the data to each DSN. This process iterates until all the data is transmitted. If any of the DSNs detect a disk error, then the done message sent to the source indicates the logical disk in error and the block where the error occurred.

4 Parallel I/O

This section describes parallel I/O in general. The current implementation, described below, uses the time sharing daemon to dispatch a user's parallel I/O request. Scheduling constraints dictated that the time sharing daemon handle parallel I/O. Our eventual goal is to move parallel I/O completely into the kernel and remove some of the latency and overhead. There are several other issues, such as supervisor access to the Control Network, that make this work more than trivial.

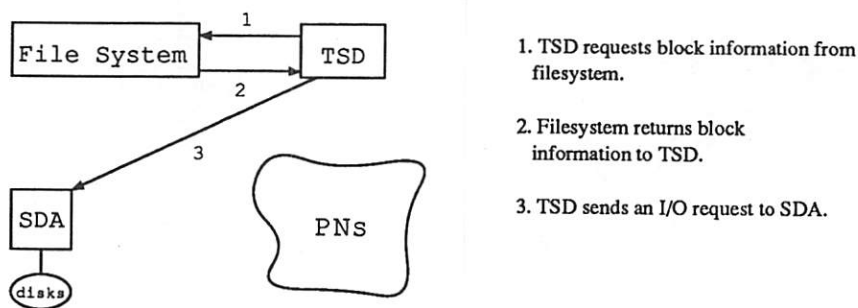


Figure 4: Beginning of Parallel I/O

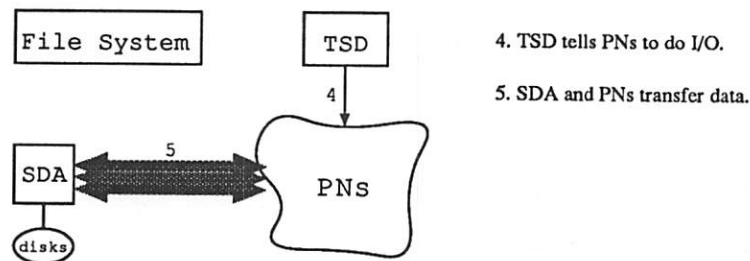


Figure 5: Transferring the Data

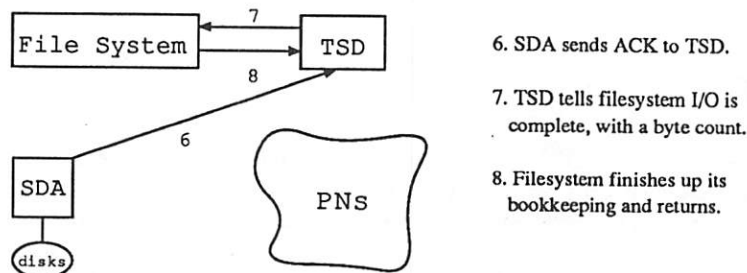


Figure 6: Finishing the I/O

A parallel I/O request really contains three distinct operations. The first, illustrated in Figure 4, shows the time sharing daemon asking the file system for the list of disk blocks relevant to the I/O request. The second portion, shown in Figure 5, indicates that once the I/O is set up, the time sharing daemon tells the PNs to commence the I/O, and a high bandwidth transfer takes place directly between the SDA and the PNs. The final operation, seen in Figure 6, cleans up the remainder of the I/O. The SDA sends an acknowledgment to the time sharing daemon, who then finishes the I/O with the file system. The file system unlocks the inode, increments the file offset and cleans up from the I/O.

The major drawback in this scheme is the latency involved in initiating a parallel I/O operation. This overhead is due to the setup time involved in passing the *ts-daemon* the user's file descriptor and to the time spent by the *ts-daemon* retrieving the block information from the kernel and then coordinating the transfer between the SCN's and the processing nodes. Once the data starts flowing between the SDA and the processing nodes, the disks perform quite reasonably as described in Chapter 7.

5 Scalable File System

Since we began with a goal of a UNIX-compatible file system, it seemed logical to begin with the UFS sources and modify them to suit the needs of our system. Basically, four major areas needed support and modification in our new file system:

1. Support for large files.
2. Support for our parallel read and write calls.
3. Handling of file system block sizes that are not a power of two.
4. Slight modifications to block allocation.

Our intent was to preserve the current performance of UFS and not impact the standard read or write paths with our changes. We can boot an *sfs* file system on a Sun workstation on a SCSI disk. Timing tests show that our file system performs the same as UFS, when running to the same disk. While not surprising, it was pleasing to know we had no negative impact on general performance in the file system.

5.1 Large Files

Supercomputer applications typically deal with very large files. Data sets of tens of gigabytes are not uncommon. Therefore, file system support for very large files is imperative. While the Berkeley Fast File System itself does nothing to restrict file sizes, higher level file I/O support within SunOS limits file sizes and offsets to sizes representable within 32 bits.

In order to support very large files, several changes were required to SunOS above the file system level. One change was to add a *vfs* flag, *VFS_BIGFILES*, indicating that the file system supports large files. Other changes relate to maintenance of position within a file, and to representation of the absolute size of a file. We address file offset maintenance first.

In order to keep track of file offsets not representable in 32 bits, we added a new fundamental system data type which we called a *longlong_t*. This structure is a composite type of two 32

bit integers laid out most significant word first, for compatibility with the Gnu C compiler GCC. An additional type, `offset_t`, is defined simply as a `longlong_t` for clarity when dealing with file offsets.

By including an ANSI `long long` within the type, user space applications including kernel header files can be compiled with an ANSI compiler, allowing them to use long long arithmetic directly. We did not feel comfortable switching the compiler used to build the UNIX kernel itself at this point, so kernel operations on `longlong_t` data structures are handled explicitly in the code.

Only two system data structures required changes to support large file offsets. First, the file offset field within the file structure was changed from an `off_t` to an `offset_t`. The `uio_offset` field within `struct uio` was also increased to 64 bits. By adding macros to each modified header file defining the old structure element names to refer to the low order word of the new 64 bit offset fields we could minimize the changes required to the system outside the system call layer and the SDA file system itself.

In addition to extending the data structures, minor changes were required to several routines at the file and vnode layers of the kernel. Any routines above the file system layer which manipulated file offsets needed to be taught about the new 64 bit types. Specific examples of code requiring slight changes include `uiomove()` and `seek()`. We also added a new `llseek()` system call to allow programs to manipulate file position beyond 2 Gigabytes.

Slight changes to the core system were also necessary to handle very large files. In SunOS, file system attribute information is contained in a `struct vattr`, which carries `stat` information in a file system independent format. The file size component of this structure was extended to 64 bits. As with file offsets, we took care here to allow existing kernel code to continue to access the low order portion of the file size field with the same name, eliminating the need for extensive changes within non-*sfs* file system code.

Special enhancements for *sfs* were required in performing serial I/O when the current seek position was not representable in 32 bits. SunOS performs serial I/O operations through the VM system. The high level file system entrypoints (the *VOP_RDWR* routines) perform any necessary block allocation, and then copy data between user space and a kernel virtual memory segment which is backed by the file system. For each page within this segment, the VM system tracks the inode and file offset mapped by this page. Unfortunately, this file offset is stored as a 32 bit quantity. Extending this field to 64 bits required much more extensive surgery on the SunOS VM system than we cared to undertake. Instead, we chose to handle serial file I/O differently for offsets requiring 64 bit representation.

The *sfs* serial I/O routine (*SFS_rwip*) handles serial I/O for file offsets up to 2Gb exactly as is done within the SunOS UFS file system. By following this path we can support mapped files and text paging within the *sfs* file system just as within UFS, as long as the file offset remains representable within 32 bits. For large files a different path is taken. I/O is done through the buffer cache rather than through the VM system. The buffer cache tracks blocks by their block number in `DEV_BSIZE` (512 byte) units. By maintaining the cache by block number we effectively get 8 additional bits of file offset maintenance in the cache, allowing *sfs* to support read and write operations on file offsets up to $2^{39}-1$. Mmap operations on files at offsets not representable in 32 bits are not supported.

5.2 Parallel I/O

One of the key features of the SDA File System is its ability to support data parallel I/O operations. Parallel I/O is implemented as a combination of code within the time sharing daemon, and file

system support code. The file system provides the mapping between file name and block lists, and handles all serial I/O. The timesharing daemon actually performs all parallel I/O operations.

Two new system calls were added to support this function. `CM_read_raw()` and `CM_write_raw()` are very similar in that both calls return lists of blocks to the time sharing daemon, with only `CM_write_raw()` performing block allocation. Each of these calls is executable by the time sharing daemon process only.

The time sharing daemon performs parallel I/O operations on behalf of user programs. Using a file descriptor passed from the application, the time sharing daemon calls the *sfs* through the `CM_read_raw()` and `CM_write_raw()` system calls to obtain a list of blocks involved in the I/O operation.

The first call to `CM_read_raw()` or `CM_write_raw()` places a lock on the inode involved. This lock is held for the duration of the entire parallel I/O operation, being released only when the timesharing daemon finishes the parallel I/O by calling `CM_read_raw()` or `CM_write_raw()` again.

A parallel I/O operation generally involves a large amount of data. As it uses the existing UFS on-disk structure, the *sfs* represents files as lists of individual blocks. But unlike standard UFS disk drivers, the SDA is optimized for transferring large contiguous block ranges, not for individual block reads or writes. The `CM_read_raw()` and `CM_write_raw()` system calls support this by coalescing the blocks returned by the file system block lookup routines into lists. The block lists returned to the time sharing daemon are thus effectively a set of extents to be used in performing the I/O operations[6].

5.3 Odd File System Block Sizes

We use the term *block stripe* to refer to one 512 byte block on each data disk. The term *disk stripe* is used to describe a logical chunk of data spread evenly with 16 bytes of data on each disk in the array. Therefore, the block stripe size of a 46 data drive logical device is 23 Kbytes, and the disk stripe size of such a configuration is 736 bytes.

UFS was designed for disk systems where the file system logical blocksize is some power of two multiple of the fundamental disk block size. With the *sfs* we were faced with the situation where the fundamental disk block size could be substantially larger than the desired logical file system blocksize, and could under many conditions bear no reasonable mathematical relationship to it whatsoever. A number of changes were required to support this.

Adding support for odd block sizes involved changes in two areas, file system macros and serial read and write processing. First all the file system macros which converted bytes to blocks, blocks to fragments, etc required modification to use division and modulus instead of shifts and masks. These changes were fairly straightforward and didn't involve major changes to the file system code itself. In benchmarking the system on a Sun 4/300 with a SCSI disk we found that this had little or no impact on file system performance.

Enhancing the serial I/O routines to handle odd block sizes required more effort. As mentioned earlier, SunOS performs serial I/O operations through the VM system. The high level file system entrypoints (the *VOP_RDWR* routines) perform any necessary block allocation, and then copy data between user space and a kernel virtual memory segment which is backed by the file system. The copy causes a fault, resulting in the file system page handling routines (*VOP_GETPAGE* or *VOP_PUTPAGE*) being called to perform the actual I/O. But by this time the I/O operation has been broken up into some sequence of *PAGESIZE* units, regardless of any effort by the user application to do I/O in file system blocksize units!

Pages do not map nicely onto file system blocks which are not a power-of-two number of bytes in size. The challenge in supporting serial I/O efficiently with odd block sizes is to group page I/O's together to form some integral number of full block I/O's. The *sfs* *getpage* and *putpage* routines attempt to do disk I/O in chunks of the least common multiple of the machine pagesize and the file system blocksize. A second problem is that a single page can span several logical blocks within a file. A single page may require two separate disk I/O's to complete. By limiting the minimum file system blocksize to be at least as large as the system pagesize we avoid the situation where three I/O's may be required to handle a page.

5.4 Block Allocation

The *sfs* uses the UFS on disk structure and allocation policies. When rotational delay is zero, the UFS policies reduce approximately to a cylinder group based extent allocator. The file system will attempt to allocate new blocks immediately following the last block in an existing file. The UFS policies generally allocate blocks for different files from different cylinder groups, with the exception of files within the same directory, which are allocated together.

This policy, in combination with extent coalescing in the parallel I/O routines, suits *sfs* fairly well. Problems arise, however, when simultaneous file extensions are done to files being allocated within the same cylinder group. Under this situation the files are not allocated as extents but instead have their blocks intimately mingled together. For parallel I/O operations this results in a single I/O requiring many extents, greatly reducing performance.

Although work in this area is not yet complete, an approach involving applying an allocation lock on a cylinder group basis for the duration of a file extension operation has shown some potential for increasing the average size of extents allocated under these conditions.

6 Network File System

The CM-5 has several SPARCstations acting as partition managers. Therefore, it seemed natural to have partition managers mount *sfs* file systems over NFS[8]. Several reasons motivated this choice over implementing our own mechanism for remotely accessing the file system. Foremost, NFS is a widely used and well understood method. The extensions necessary to support our parallel calls would be minimal. Also, since *sfs* is UNIX-compatible, users on any workstation could remotely mount and access the SDA.

At mount time, the partition manager, as a client, determines whether the remote file system is a CM-5 file system. This information is maintained in a bit in the *mntinfo* structure. If the bit is set, the parallel I/O system calls are allowed. Otherwise an error condition is returned to the user.

Two new RPC calls were added to support the *CM_read_raw()* and *CM_write_raw()* parallel I/O system calls from the time sharing daemon. Since the parallel I/O calls split into a "begin" and an "end" half, the new RPCs, *RFS_PARIOB* and *RFS_PARIOE*, are functionally split along those same lines. The first returns the disk block information to the time sharing daemon, and the second completes the operation. The same locking issues that are present in *sfs* apply here as well for the *rnode*. The *rnode* is locked when the "begin" call completes, and it does not become unlocked until the "end" call.

Additional RPC calls were necessary to handle file sizes greater than 2 gigabytes due to in-

creased sizes in some of the message fields. We needed to support 64-bit size fields while maintaining compatibility with existing NFS protocols. Five calls required these changes:

1. Getting attributes, *RFS_GETATTRL*.
2. Setting attributes, *RFS_SETATTRL*.
3. Reading, *RFS_READL*.
4. Writing, *RFS_WRITE_L*.
5. Lookup operations, *RFS_LOOKUPL*.

We could not change the existing versions of those calls, because then our NFS would not be compatible with the rest of the world for standard UNIX calls. So, we added new 64-bit versions of those calls. The need for 64-bit version of the first four calls is straightforward. Lookups needed to support 64-bit files because the result of a lookup operation returns attributes about the file it looked up, including the file size. As mentioned in Section 5, each mount point may have a flag set indicating it supports "big" files. If that flag is set, then the client uses the 64-bit versions of those calls where appropriate.

7 Performance

Our goal was to provide a scalable file system capable of running on top of the SDA within a CM-5 system. The performance goal for the first release of the *sfs* software was to obtain a rate of 1.5 Mb/second per disk on read operations and a rate of 1 Mb/second per disk on write operations across a wide range of disk configurations. We have achieved these performance goals on file systems spanning from 16 to 118 disk drives. Although a system with 118 data drives was the largest we had available to run the tests, the hardware and software systems can support much larger configurations.

Our test measured the amount of time it took to execute the parallel read and write system calls. Each performance test was run ten times for each data point. The highest and lowest performance result were discarded and the other eight trials were averaged and plotted here. All the performance numbers presented in this paper were measured reading and writing regular files, not using the raw device. Each test was run on a partition with 128 nodes. The system was up and running in multi-user mode, although our program was the only parallel process running.

A logical device generally contains data drives and one parity drive and one spare drive to use in case of a disk failure. Therefore one to five backplanes of SDAs typically contain 22, 46, 70, 94 and 118 data disks respectively. We also show performance numbers for power of two data disk configurations containing 16 and 32 data disks. The disks are SCSI based IBM Model 0663E15, which are able to sustain raw transfers at approximately 2 Mb/second on reads and about 1.8 Mb/second on writes.

Figure 7 shows how the effective transfer rate for read operations varies according to the size of the transfer for file systems utilizing various disk configurations. The graph shows that read performance improves as we continue to add disks drives over a range from 16 to 118 drives. The highest read transfer rate achieved was over 185 Mb/second when using 118 data drives. Each curve periodically shows a slight drop in performance at transfer sizes that are somewhere between 64 and 256 Mb. This slight decrease in performance at those specific values is due to the additional overhead involved in switching to a different indirect block. File systems with a power of 2 number

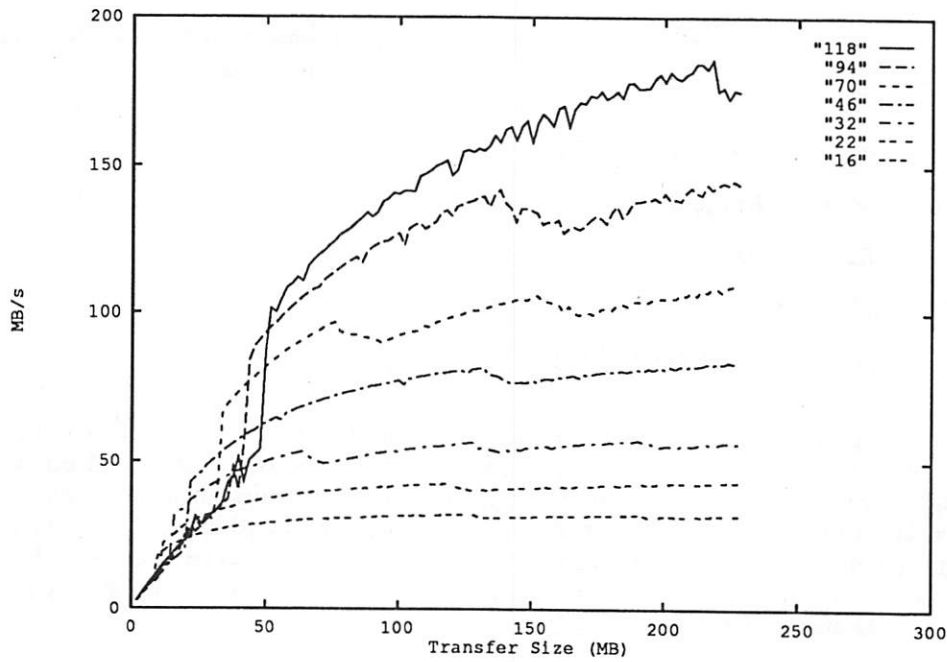


Figure 7: Read performance with 128 nodes

of disks have a file system block size of 16Kb. This allows for 4096 block addresses in each indirect block. Each indirect block can thus address $16\text{Kb} \times 4096 = 64\text{Mb}$ of data. This is most clearly shown in the line for 32 disk drives. For the file systems with a non power of two number of disks, the file system block size is somewhere between 16Kb and 32Kb depending on the number of disks. Notice that for small transfers, larger disk configurations perform more slowly. This is shown by the crossing lines of the graphs near the origin. This behavior reflects additional latency in the larger configurations, due to their having additional DSN controllers and hence, needing to send more messages. The next release of the system software will decrease this latency with the goal of eliminating any differences in performance at the small transfer sizes.

Figure 8 shows how the effective transfer rate for write operations varies according to the size of the transfer for file systems utilizing various disk configurations. Results are presented for the same disk configurations as reads. The shapes of the curves are similar between reads and writes with the write performance topping out around 1 Mb/second per disk while the read performance tops out around 1.5 Mb/second per disk. Write performance is less than read performance for a number of reasons, including the fact that the physical disks perform reads faster than writes and also the buffer size limitations on the DSN boards. Again notice the slight dips due to indirect block access. Latency for large disk configurations causes the lines to cross near the origin for writes as well.

Figure 9 shows how the maximum effective transfer rate **per disk** for read and write operations varies according to the number of disks used for file systems with a varying number of drives. The amount of data transferred per disk is held constant at 2 Mb for this graph. (I.e. 32Mb for 16 disks, 44Mb for 22 disks, etc.) An optimum line for this graph would be a perfectly flat horizontal line. Our data shows nearly optimum scaling. This graph clearly shows that we have achieved scalable file system performance across almost an order of magnitude change in the number of disk drives.

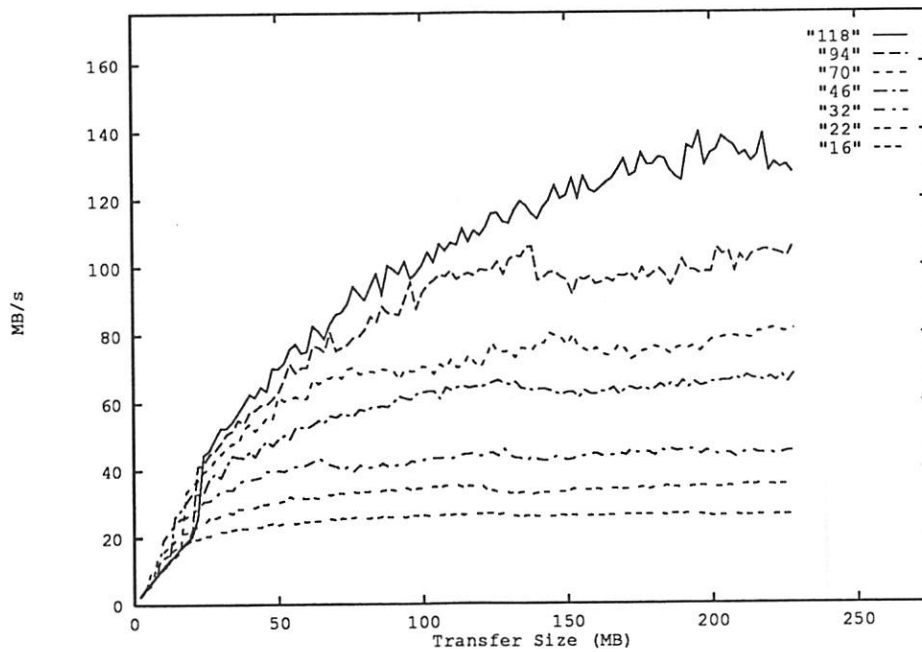


Figure 8: Write performance with 128 nodes

The performance of the SDA file system is determined by several factors, including the time required to execute the file system code on the CP and the time required to process the request on the DSN controllers. These times remain relatively constant regardless of the partition size and disk configuration. The number of disks used and the number of PNs in the partition determine the effective transfer rate, which allows the performance of the file system to scale according to the size of the machine. By allocating blocks contiguously we are able to sustain transfer rates which are a significant percentage of the raw disk transfer rate.

8 Conclusions and Future Work

We have shown that you can build a UNIX-compatible file system which is truly scalable in both size and performance. For large data transfers 118 disks have been used in parallel to achieve extremely high transfer rates on a single I/O. Modifying the disk block allocation algorithms to give us large contiguous extents gave us excellent performance results while not adversely affecting the standard read and write paths. We have shown that we have met our performance goals for reads and writes.

We recognize some limitations and latency issues in our implementation. We would like to move the parallel I/O system calls into the kernel and remove the overhead of having the time sharing daemon coordinate the I/O. We would like to address the latencies involved in coordinating many DSN boards. Obviously, we are also investigating new hardware, faster and bigger disk drives to increase the performance of the system.

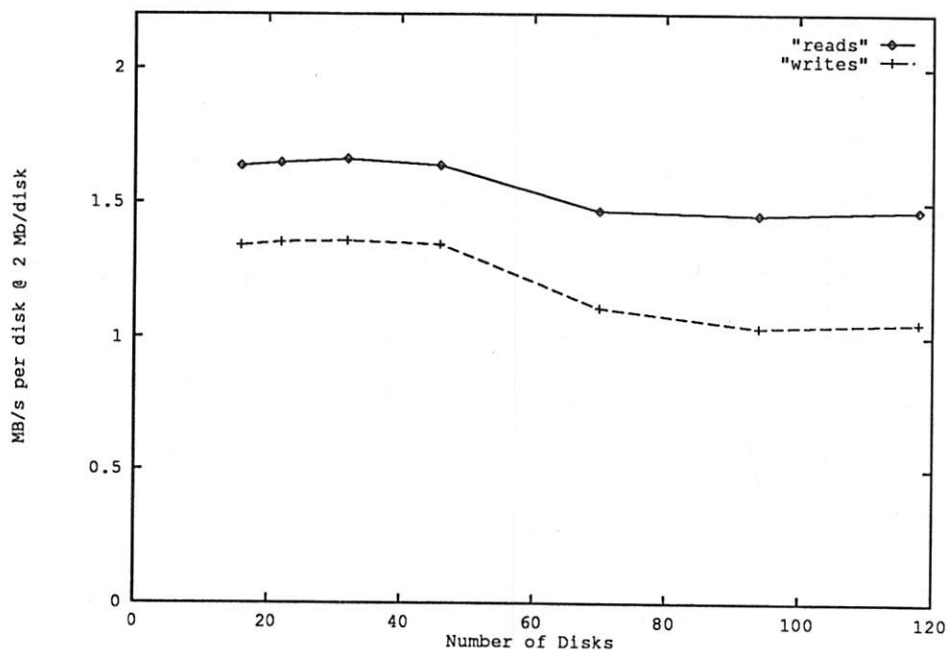


Figure 9: Per-disk read and write performance with 128 nodes

9 Acknowledgments

We would like to thank John Smith, Tom Moser, Eric Rowe, Soroush Shakib, Eric Sharakan, Andre Vignos, Roger Lee, Mark Bromley and David Taylor. Their help and contributions to this project were key to its success.

References

- [1] S. Coleman and editors S. Miller. Mass Storage System Reference Model: Version 4. IEEE Technical Committee on Mass Storage System and Technology, May 1990.
- [2] B. Kahle, W. Nesheim, and M. Isman. Unix and the Connection Machine Operating System. In *Workshop Proceedings, USENIX Workshop on Unix and Supercomputers*, pages 93-107, Pittsburgh, PA, 1988. USENIX Association.
- [3] S. Leffler, M. K. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [4] S. LoVerso, N. Pacioret, A. Langerman, and G. Feinberg. The OSF/1 Unix Filesystem (UFS). In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 207-218, El Toro, CA, 1991. USENIX Association.
- [5] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2:181-197, August 1984.
- [6] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 33-43, El Toro, CA, 1991. USENIX Association.
- [7] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Conference Proceedings, 1988 SIGMOD Conference*, pages 109-116. Association of Computing Machinery (ACM), 1988.

- [8] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon. Design and Implementation of the Sun Network Filesystem. In *Conference Proceedings, 1985 Summer USENIX Technical Conference*, pages 119–130, Berkeley, CA, 1985. USENIX Association.
- [9] M. Schulze. Considerations in the Design of a RAID Prototype. In *Master's Report*, Berkeley, CA, 1988. University of California at Berkeley.
- [10] Thinking Machines Corporation, Cambridge, Massachusetts 02142-1264. *The Connection Machine CM-5 Technical Summary*, 1991.

10 Author Information

Sue LoVerso is a senior software engineer at Thinking Machines Corporation working on the CM-5 Operating System. Prior to joining Thinking Machines, she was employed at Encore Computer Corporation for several years as a member of the Mach Operating System group. She received her Master's degree in Computer Science from the State University of New York at Buffalo. Sue is a member of the IEEE Computer Society and the Society of Women Engineers. She can be contacted at sue@think.com.

Marshall Isman has been with Thinking Machines Corporation since 1986 and is a manager of Operating Systems. He has been involved in the architecture, design and implementation of the CM2 and CM5 I/O systems and Operating Systems. Prior to joining Thinking Machines Corporation, Marshall worked at Computer Consoles and Bell Telephone Laboratories. Marshall holds a BS from State University of New York at Albany and an MS from Harvard University in Computer Science. He can be contacted at marshall@think.com.

Andy Nanopoulos is a senior software engineer at Thinking Machines Corporation and was responsible for the design and implementation of the DSN software on the SDA. He has a BS in Computer Engineering from Boston University. Before coming to TMC he was president of Yorfrenz Development Corporation. He can be contacted at andyn@think.com.

Bill Nesheim holds a BS from Cornell University. He has been with Thinking Machines since 1986, where he is currently manager of I/O system software. Prior to joining Thinking Machines he was a research programmer at the Cornell University Computer Science department. He can be contacted at nesheim@think.com.

Ewan Milne is an OS Engineer working on CMOST development. He is currently working on the SDA project. Prior to joining Thinking Machines in October of 1992, he worked for 8 years at Prime Computer Inc., where he was a Principal Engineer in the OS group. Ewan studied Computer Science at Rensselaer Polytechnic Institute and is currently completing his Bachelor's Degree at Boston University. He can be contacted at milne@think.com.

Rick Wheeler has a BA from Brandeis University and a MSc from Hebrew University. Before joining Thinking Machines in 1990, he was a member of the staff at the Israel Air Force Institute and Hebrew University's Computer Science Department. He can be contacted at ric@think.com.

Adaptive Block Rearrangement Under UNIX*

Sedat Akyürek & Kenneth Salem
Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract

An adaptive UNIX disk device driver is described. The driver copies frequently-referenced blocks from their original locations to reserved space near the center of the disk to reduce seek times. Reference frequencies need not be known in advance. Instead, they are estimated by monitoring the stream of arriving requests. Measurements show that the adaptive driver reduces seek times by more than half, and improves response times significantly.

1 Introduction

In a recent paper [Akyurek 93] we introduced an adaptive block rearrangement technique which reduces disk response times by reducing seek times. In this technique, a small number of frequently referenced blocks are copied from their original locations to reserved space near the middle of the disk. The term "adaptive" means that no advance knowledge of the frequency of reference to the data blocks is required. Instead, reference frequencies are estimated by monitoring the stream of requests for data from the disk. The set of blocks in the reserved space is changed periodically to adapt to changing access patterns.

Trace-driven simulations have shown that this technique can be very effective in reducing seek times. This paper describes an implementation of the technique using a modified UNIX device driver. It also presents the results of detailed measurements taken during the operation of the system. Our results demonstrate that adaptive block rearrangement can be easily and effectively integrated under existing file systems.

1.1 Related Work

That a disk's performance can be improved by clustering frequently accessed data is well-known. If data references are derived from an independent random process with a known, fixed distribution, it has been shown that the *organ pipe* heuristic places the data optimally [Wong 80, Grossman 73]. The organ pipe heuristic calls for the most frequently accessed data to be placed in the center of the disk. The next most frequently accessed data is placed to either side of the center, and the process continues until the least-accessed data has been placed at the edge of the disk. More recently, similar results have been shown for optical storage media [Ford 91].

In practice, data references are not drawn from a fixed distribution, nor are they independent. Although references are highly skewed [Floyd 89, Staelin 91, Vongsath 90, Ouster 85], request distributions change over time, and they are generally not known in advance. Nevertheless, variations of the organ pipe heuristic seem to work well in practice. Recently, several papers have proposed adaptive applications of data clustering based on this idea.

This work was supported by National Science Foundation Grant No: CCR-8908898 and in part by CESDIS.

*UNIX is a trademark of ATT

Vongsathorn and Carson [Vongsath 90] showed that dynamically clustering frequently accessed data worked better than static placement. In that study, disk cylinders are dynamically rearranged using the organ pipe heuristic, according to observed data access frequencies. Recent work in the DataMesh project [Ruemmler 91] considered rearrangement of cylinders and blocks, with mixed results. Their conclusion that block shuffling generally outperforms cylinder shuffling corroborates one of our own. A similar approach is employed in the experimental iPcress file system [Staelin 91], which monitors access to files and moves files with high "temperatures" (frequency of access divided by file size) to the center of the disk.

Our technique differs from each of the techniques mentioned above in at least one of the following aspects.

Granularity Our technique moves blocks rather than cylinders or files. Blocks within a file or within a cylinder can vary in temperature. Also, block rearrangement can increase the number of zero-length seeks (i.e. need to seek at all), while cylinder reorganization cannot. Smaller granularity also facilitates incremental rearrangement.

Data Volume Only a small fraction of blocks are rearranged at any time as opposed to reorganizing all the data on the disk. This makes rearrangement faster.

Layout Preservation Block relocation is temporary, and cooled blocks are returned to their original locations.

Transparency Our technique can be implemented in a device driver (or controller). No changes to the file system are required.

Other systems cluster data based on criteria other than reference frequency. The Berkeley Fast File System (FFS) [McKusick 84] used in many UNIX systems uses placement heuristics that try to cluster the blocks of a file. However, hot blocks from different files may be spread widely over the disk's surface. This can result in long random seek operations when requests for the blocks of different files are interleaved, as is the case in multi-user systems.

LFS [Rosenblum 91] and Loge [English 92] systems rearrange data based on the order of writes in the request stream. The primary goal of these systems is to improve write performance, not read performance. In contrast to both Loge and LFS, our adaptive block rearrangement technique makes both read and write operations faster. These systems try to improve write performance by reducing both seek and rotational delays. LFS eliminates seek and rotational delays by combining many write operations into a single large write operation. The Loge self-organizing disk controller transparently reorganizes blocks each time they are written to reduce seek and rotational delay. Simulation studies of the controller show that it can reduce write service times, but the savings come at the expense of increased read service times. Unlike Loge, the block rearrangement system described here preserves the data placement done by the file system. Finally, although the adaptive block rearrangement system described here is implemented in the device driver on the host, it can be implemented in an intelligent IO or disk controller like Loge.

In the next section we will give an overview of the block rearrangement technique. Section 3 describes the implementation of the adaptive block rearrangement in a UNIX system. In section 4 we present the results of performance measurement experiments conducted while the system was operational on a network file server.

2 Overview

The idea of block rearrangement is motivated by the fact that access to data stored on disks is highly skewed. Of the thousands of blocks stored on a disk, a small fraction of them absorbs most of the requests. If the hot (frequently accessed) blocks are spread over the surface of the disk, distant from each other, long seek delays may result.

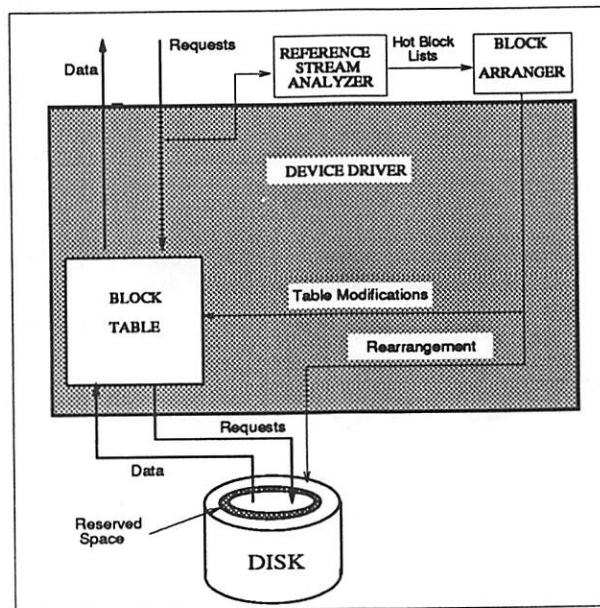


Figure 1: Adaptive Device Driver

Hot blocks can be clustered on a reserved set of contiguous disk cylinders to reduce the seek times. Under our block rearrangement technique, this is achieved by copying hot blocks on to a set of reserved cylinders in the middle of the disk. The block layout outside of the reserved cylinders is left undisturbed. Incoming requests are redirected to the reserved cylinders if the block resides there. If the reserved cylinders accommodate blocks which get most of the references, we expect that the disk head will tend to linger over those cylinders and seek distances will be reduced.

Figure 1 illustrates the organization of an adaptive block rearrangement system implemented in a disk driver. The system monitors the stream of requests directed to the disk and periodically produces a list of hot (frequently-referenced) blocks, ordered by frequency of reference. The hot block list is used to determine which blocks should be placed in the reserved cylinders. The rearrangement system copies the selected hot blocks from their original locations to the reserved cylinders. These blocks remain in the reserved space until the next hot block list is produced. Blocks which are not hot anymore are copied back to their original locations.

The block rearrangement system assigns hot blocks to the reserved cylinders according to their rank in the list, i.e., their frequency of reference, using the organ pipe heuristic. Assuming that C blocks fit on a cylinder, the C hottest blocks are placed on the middle cylinder of the reserved cylinder group. The next hottest are placed on an adjacent cylinder, and so on so that the final cylinder reference distribution across the reserved cylinders forms an organ pipe distribution.

3 Implementation

In this section we will present the implementation details of the components shown in Figure 1. The block rearrangement system is implemented through modifications to the disk device driver of SunOS[†] 4.1.1. In addition, several user level programs are used to control the modified driver.

[†]SunOS is a trademark of Sun Microsystems, Inc.

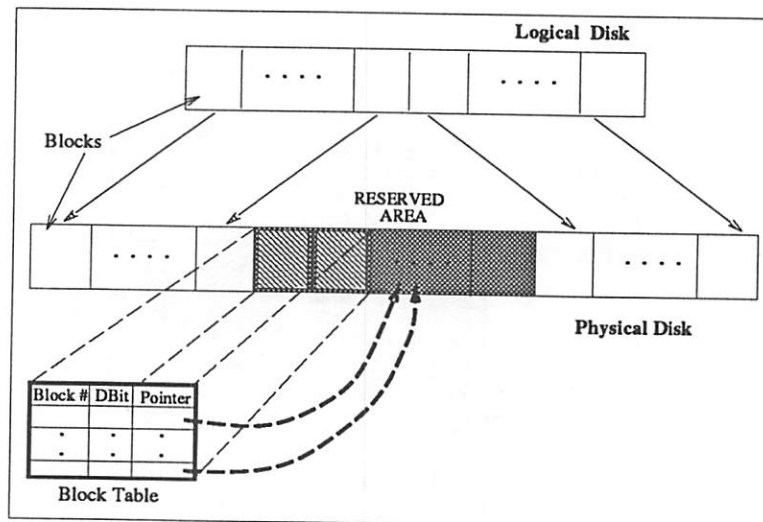


Figure 2: Block Mapping on a Rearranged Disk

3.1 Modifications in the Device Driver

The device driver modifications implement the reserved space on the disk and the mapping of hot blocks to their new positions in the reserved space. They also provide entry points to the kernel for controlling the movement of blocks to and from the reserved area and for monitoring block accesses. Another function of the modified driver is to gather performance statistics.

3.1.1 Reserved Space

Disk labels contain information about the size of the disk and the sizes and positions of disk partitions. This information is used by the `newfs` program to initialize a file system on a given partition on the disk.

To make space for the rearranged blocks, the target disk is made look smaller than it really is by changing the disk geometry information on the disk label. Disk partitioning information is also changed accordingly so that the file system thinks that the disk has fewer cylinders. The hidden cylinders implement the reserved space. The driver implements the mapping between the *logical* (smaller) disk and the actual disk (Fig. 2)[†].

When a target disk is initialized for rearrangement, the first sector and the length of the reserved space are recorded in its label. During initialization a special value is also recorded in the label to mark it as a "rearranged" disk[‡]. At the time of system start-up, the `attach` routine of the driver checks if the disk is a rearranged disk. If it is, then the information about the reserved area is read in to be used for block mapping.

3.1.2 Block Mapping

The `strategy` routine of a disk driver is responsible for converting file system's block addresses to physical block addresses. In the modified driver, it also does the mapping from the *logical* disk to the real disk and

[†]In SCSI disks the disk interface presents the disk space as a sequence of logical sectors. The actual physical locations of the sectors are not known. We rely on an implicit assumption that most SCSI sector numbers are close to their true physical numbers.

[‡]Block rearrangement is applied on a physical device basis. That is, a disk may have several partitions and consequently several file systems on it, but all the blocks on the disk can be rearranged regardless of which partition they belong to. We require each file system on the disk to have the same block size.

reroutes requests for rearranged blocks when necessary.

When a request arrives for a rearranged disk, the **strategy** routine first converts the logical block address to a physical block address. It then determines whether the block has been repositioned into the reserved area. To implement this check the driver maintains a data structure called the *block table* for the rearranged blocks. When a block is copied into the reserved space, it is entered to the table. The table records the block's old and new physical addresses and a dirty bit. If an entry for the requested block is found in the block table, its new physical address is used to retrieve the data.

A copy of the block table is also stored on the disk (at the beginning of the reserved area) to be used at start-up and for recovery purposes. The **attach** routine of the driver reads in the block table at start-up. The copy of the block table on the disk always correctly reflects the rearranged blocks and their positions in the reserved area (unless media failure occurs). However, the dirty bit information may not always be up-to-date. Dirty bits are used to determine if rearranged blocks have been written. They determine whether blocks must be copied back to their original locations when they cool down. In our implementation, if there are repositioned blocks in the reserved area at the time of start-up, they are all marked as dirty. This conservative strategy insures correct operation in case a repositioned block is written and the system operation is interrupted before this was reflected on the disk copy of the block table.

The size of a "block" in the rearrangement system is the size of a file system block. The file system requests at most one block of data from the disk at a time. The raw disk IO interface on the other hand, allows requests larger than the block size to be forwarded to the disk. Driver routines responsible for raw IO were changed to break large requests into block-sized subrequests.

3.1.3 Block Movement

The driver provides kernel entry points through **ioctl** calls for user-level processes to control the movement of blocks into and out of the reserved area. It implements two **ioctl** calls for block movement :

DKIOBCOPY copies a given block to a given address in the reserved area. It also places an entry for the block in the *block table* and forces *block table* to be written to disk.

DKIOCCLEAN cleans the reserved area by removing the blocks from the *block table* one at a time. Blocks leaving the reserved area are copied to their original disk positions if they have been updated, i.e. if their dirty bit is set. The *block table* is written to disk after each block is moved out.

Copying a block into the reserved area requires three IO operations. Moving a block out of the reserved area requires at least one IO operation and two extra operations if the block is dirty. However, other requests can interleave with these operations. Requests for a block that is being moved are delayed temporarily by the driver.

3.1.4 Request Monitoring

The driver records requests (block number, request size, flags etc.) in a small internal table. Request recording stops when the table is full, and resumes when the table is emptied. An **ioctl** call enables user processes to read the contents of the table and to clean it.

A user process periodically reads the contents of the request table and uses the them to analyze the block accesses. The frequency of request table reads can be changed. We have used a period of two minutes in our experiments. This was short enough to capture almost all the requests.

3.1.5 Performance Monitoring

Another function of the driver is to gather statistics about different measures related to disk accesses, such as seek distances, service times and queueing times. This is provided for monitoring the performance of the system and is not a part of the block rearrangement system.

Performance monitoring is implemented much like request monitoring. Statistics are recorded in a table inside the driver. User-level processes can read the contents of this table through an `ioctl` call which also resets the values in the table. The table contains statistics collected since the last time the table was read. The beginning and ending times of the recording are also entered into the table.

All statistics are recorded separately for read operations and write operations. The table records seek distance distributions (in arrival order and in scheduled order) and service time and queueing time histograms. Time histograms are recorded with a resolution of 1 ms. Cumulative service times and queueing times are also recorded. Queueing time for a request is the period between the time the driver first receives the request and the time the request is submitted to the disk. Service time for a request is from the end of the queueing time to the time the request is returned from the disk.

3.2 User Level Programs

User level programs use the entry points provided by the modified kernel to monitor and analyze block access frequencies. Using this information, they decide which blocks should be rearranged and where they should be placed in the reserved area. One user-level process, the *reference stream analyzer*, monitors the block accesses and tries to determine the frequently accessed blocks and their access frequencies. Using this information, another process, which is called the *block arranger* decides which blocks are to be rearranged. This process also controls the placement of the selected blocks in the reserved area.

The functions implemented by the user-level processes can easily be incorporated in to the device driver itself. In particular, our techniques for estimating block reference counts are fast and require little space[¶], and the block placement policies are very simple. However, we wanted to use our system as a test bed to experiment with different methods for learning access frequencies and different rearrangement policies. Implementing those functions in user-level processes gave us flexibility in our experiments.

The block arranger implements three block placement policies :

Organ-pipe placement This policy arranges the blocks chosen for rearrangement in an organ-pipe pattern according to their access frequencies. This done by placing the blocks with highest access frequencies in the center cylinder of the reserved area and continuing to fill cylinders on alternating sides of the center with the blocks which have the next highest access frequencies.

Interleaved placement SunOS UNIX file system (UFS), which is based on FFS, places the blocks of a file interleaved by an interleaving factor. The purpose is to reduce rotational delays while accessing a file sequentially. Our interleaved placement policy tries to preserve the interleaving performed by the file system. This policy begins by placing the hottest block first, like the organ-pipe placement. However, after it places a block it does not immediately select the next hottest block for placement. Instead it looks for a block which could be the previous block's successor in the file system's interleaved placement. The successor's physical location must of course succeed the previous block's physical location by the interleaving factor. The placement policy also requires the successor's access frequency to be close^{||} to the previous block's access frequency (this is the policy's criterion to decide whether two blocks belong to the same file). If such a "could be successor" block is found, it is placed using the same interleaving factor used by the file system. This process continues as long as a chain of such probable

[¶]The hot block estimation algorithm we have used and several other space-efficient hot spot detection algorithms are presented in [Salem 92].

^{||}The successor's access frequency is defined to be close if it is at least 50% of the previous block's frequency. The 50% figure was selected arbitrarily.

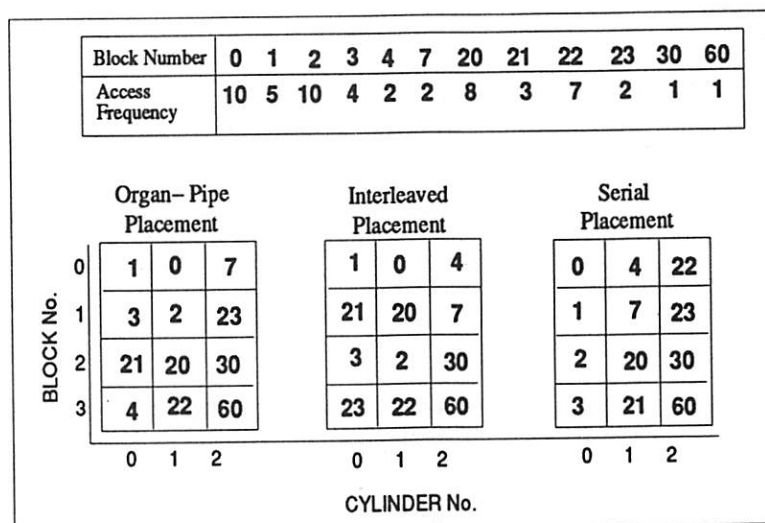


Figure 3: Placement Policies. The block placement of different policies are shown. The set of blocks to be rearranged and their estimated access frequencies are also given. The reserved area has three cylinders with four blocks in each cylinder. The interleaving factor (rotational delay) of the file system is assumed to be one block. The interleaved placement policy considers the access frequency of a block “close” to another block’s, if the first block’s frequency is at least half of the second block’s access frequency.

successor blocks are found. The interleaved placement also stops when the end of the current cylinder is reached. When the policy quits interleaved placement mode, it continues by choosing the block which has the next highest access frequency.

Serial placement This is the simplest placement policy. Blocks that are chosen for rearrangement are placed in the reserved space in ascending order of their original block numbers. This policy needs to know only the block numbers of the hot blocks. Access frequencies of the hot blocks are not necessary.

4 Performance Measurements.

In this section we will present the results of a series of experiments we have conducted to monitor the adaptive driver’s performance. We will first describe the environment in which the experiments were run.

4.1 Experimental Environment

The adaptive block rearrangement system is operational on a Sun Sparcstation** 2, called Sakarya. The operating system on the machine is SunOS 4.1.1. The file system block size is 8 kilobytes and the fragment size is 1 kilobyte.

Sakarya has a main memory of 32 megabytes. The amount of memory used for buffer cache is not fixed. Under SunOS the memory pages are used for both process pages and IO pages on demand. This means all that of the available memory can potentially be used as a buffer cache [McVoy 91]. Sakarya is used mainly by one of the authors only and is usually very lightly loaded. As a result, a large portion of the memory is available as a buffer cache most of the time.

Sakarya has two disks attached to it and one of them is used as a rearranged disk. The disk used for block rearrangement is a Toshiba Model MK156F SCSI disk. Specifications of the disk and its seek time

**Sun and Sparc are trademarks of Sun Microsystems, Inc.

Toshiba MK156F SCSI DISK	
Capacity (MB)	135
Cylinders	815
Tracks/Cyl	10
Sectors/Track	34
RPM	3600

$$seektime(d) = \begin{cases} 0 & \text{if } d = 0 \\ 6.248 + 1.393\sqrt{d} - 0.99\sqrt[3]{d} + 0.813 \ln d & \text{if } d < 315 \\ 17.503 + 0.03d & \text{if } d \geq 315 \end{cases}$$

Table 1: Specifications of the disk. Seek time is given in milliseconds as a function of seek distances (in cylinders).

function are given in Table 1. The seek time function is borrowed from [Carson], in which the disk delay parameters for this disk were measured and a precise seek time function was devised.

Out of the disk's 815 cylinders, 48 cylinders (3 cylinder groups) in the middle of the disk have been used as reserved cylinders. This amounts to approximately 8 megabytes (6% of the total disk capacity). Approximately 1000 8Kbyte-blocks can fit in this space. We chose the size of the reserved space in the light of the results of our previous studies [Akyurek 93]. The results of trace-driven simulations in that study showed that most of the benefits of block rearrangement were realized by rearranging 1%-2% of the total number of blocks. The simulation results also showed that the additional benefits of using more blocks decreased sharply.

We used Sakarya as an NFS server for 14 sparcastations for executable files and libraries. The disk stored parts of the `/usr` file system. The file system was mounted read-only on the client workstations. The user population of the workstations consisted of about 40 faculty and graduate students in the Computer Science Department of the University of Maryland.

We also experimented with a read/write mounted file system stored on the rearranged disk. In this experiment the disk stored the home directories of 10 users whose primary activities include running simulations, editing papers and electronic mailing.

4.2 Experiments

We will present the results of four sets of experiments. In the first set, performance improvements due to block rearrangement were monitored when Sakarya's disk was storing the `/usr` file system. In the next two groups of experiments we varied the number of rearranged blocks and compared the effects of different rearrangement policies, respectively. These experiments were run when Sakarya held the `/usr` file system. In the last set of experiments we used rearrangement when the disk stored the home directories.

During the operation of the system, block access information that was monitored on one day was used at the end of the day for rearranging the blocks for next day's operations. Statistics were gathered between 7am and 10pm about the system's performance. The timing measurements have a resolution of 5 microseconds on Sakarya.

4.2.1 Rearrangement of `/usr`

The block rearrangement system was operational on Sakarya for several months when it was serving 14 client workstations. We applied block rearrangement on alternate days for several weeks to assess the performance improvements. When rearrangement was applied, 1018 blocks were placed in the reserved area using the organ-pipe placement heuristic.

Results from four consecutive days are summarized in Tables 2 and 3. Results for the other days are similar. Table 2 presents the measurements for all requests whereas Table 3 describes only read operations. On days 1 and 3, block arrangement was not applied and original copies of blocks were used. On days 2 and

	Day 1	Day 2	Day 3	Day 4
Rearrangement	Not Applied	Applied	Not Applied	Applied
FCFS Mean Seek Dist (cyln)	220	225	235	228
Mean Seek Distance (cyln)	173	8	183	5
Zero-length Seeks (%)	23	88	19	92
FCFS Mean Seek Time (ms)	20.92	21.46	22.33	21.91
Mean Seek Time (ms)	18.21	1.55	19.24	0.98
Mean Service Time (ms)	38.41	22.95	39.71	22.61
Mean Waiting Time (ms)	87.30	50.03	94.51	48.82

Table 2: Experimental results for /usr (all requests).

	Day 1	Day 2	Day 3	Day 4
Rearrangement	Not Applied	Applied	Not Applied	Applied
FCFS Mean Seek Dist (cyln)	131	165	182	192
Mean Seek Distance (cyln)	123	23	170	17
Zero-length Seeks (%)	44	67	31	72
FCFS Mean Seek Time (ms)	12.97	16.14	17.24	17.91
Mean Seek Time (ms)	12.49	4.49	16.61	3.54
Mean Service Time (ms)	30.99	24.18	35.32	22.57
Mean Waiting Time (ms)	6.64	5.47	4.76	4.46

Table 3: Experimental results for /usr (read requests only).

4, block rearrangement was applied. All of the figures reported in the tables are measured values except for seek times. Seek times are computed using measured seek distance distributions and the seek time function shown in Table 2. First-come-first-served (FCFS) order seek delays are also presented together with actual seek delays. The effect of UNIX disk-head scheduling can be seen when we compare the FCFS and actual values on days 1 and 3.

Comparing the results for the “rearrangement-on” days and the “rearrangement-off” days, we see that adaptive block rearrangement drastically reduced seek distances and seek times. Service time reductions are significant but are slightly lower than the seek time reductions. This may be due to the disturbances in the file-system rotational optimizations caused by re-arranged blocks.

The system is very successful at increasing the number of zero-length seeks. This is very important in reducing seek times because of the high constant-time overhead associated with every non-zero seek. By placing frequently accessed blocks on the same cylinders, the need to reposition the disk head was often eliminated completely. This is an advantage of rearranging blocks rather than cylinders. Cylinder rearrangement cannot increase the number of zero-length seeks.

Waiting times are also reduced as a result of block rearrangement, and this further reduces the response times for disk operations. This is particularly true for write operations because write requests arrive in bursts in UNIX systems.

A notable difference between the write operations and read operations in this environment is that block access distributions for write operations are much more skewed than they are for read operations. Since write operations are concentrated onto a smaller set of blocks, write operations benefit more from block rearrangement than read operations do. This fact explains many of the differences between the results reported in Table 3 and those in Table 2.

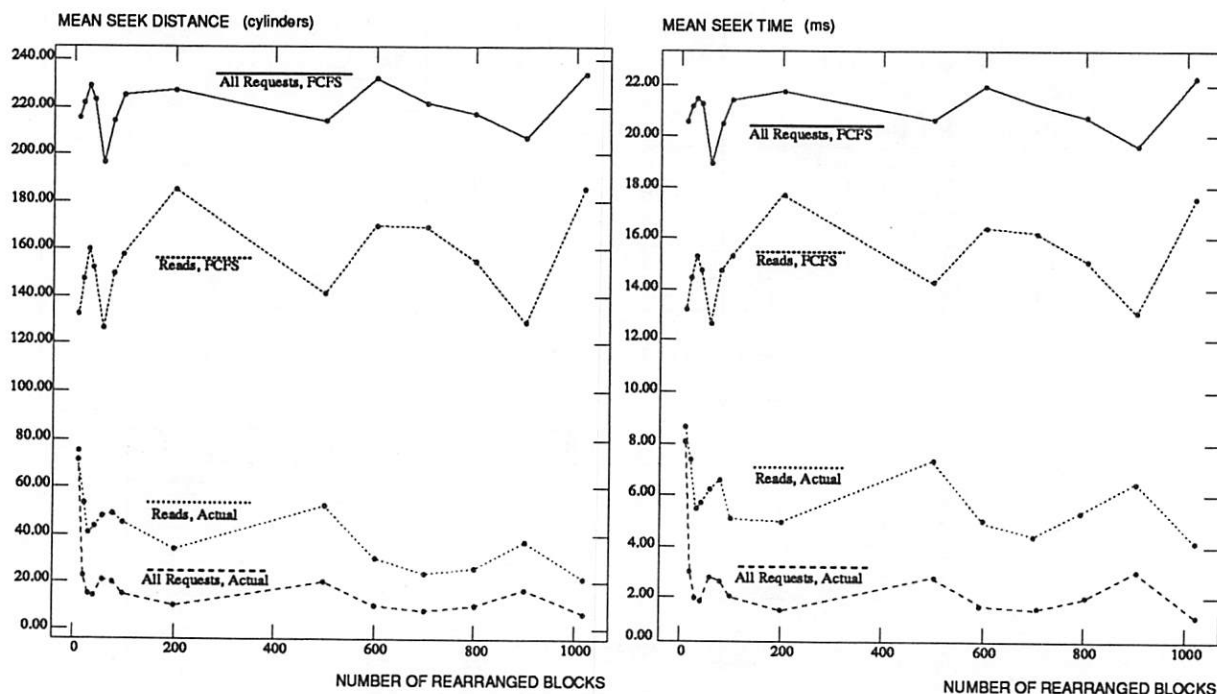


Figure 4: Results of experiments with different number of rearranged blocks.

4.2.2 Varying the Number of Rearranged Blocks

In our next set of experiments we varied the number of rearranged blocks in the reserved area. Block rearrangement was applied for several weeks (on weekdays only) by rearranging a different number of blocks each day. Results of these experiments are summarized in Figure 4. The graphs plot mean seek distances and mean seek times as function of the number of rearranged blocks. Arrival-order (FCFS) mean seek delays are also plotted. If block rearrangement had not been applied, the actual delays would have been slightly lower (due to disk head scheduling) than arrival order delays.

As the number of blocks in the reserved area is increased, we expect to see greater reductions in seek delays. However, the results of our experiments draw a different picture. Most of the benefits of block rearrangement are realized by rearranging a very small number of blocks. The marginal benefit of using additional blocks becomes very small. Rearranging blocks in excess of the hottest 200 blocks brings almost no extra benefits.

The result of these experiments can be better understood by examining the skewed block reference distribution. Figure 5 shows the cumulative distribution of block requests on a typical day. Blocks are sorted in decreasing order of reference counts. Distributions for all the requests and also just for reads are shown. Fewer than 2000 blocks absorb all of the requests. The 100 hottest blocks absorb around 90% of all the requests. Read references are also very skewed. Most of the reductions in seek delays are due to the first few hundred blocks which get most of the references.

4.2.3 Different Placement Policies

As mentioned earlier, the block arranger component of the rearrangement system implements two other block placement policies in addition to the organ-pipe placement. Although we normally used organ-pipe placement in our system, we also experimented with the other placement policies.

Tables 4 and 5 present results of experiments for the serial placement policy and the interleaved placement policy. Table 4 lists the results of a day's operation for each of the three placement policies.

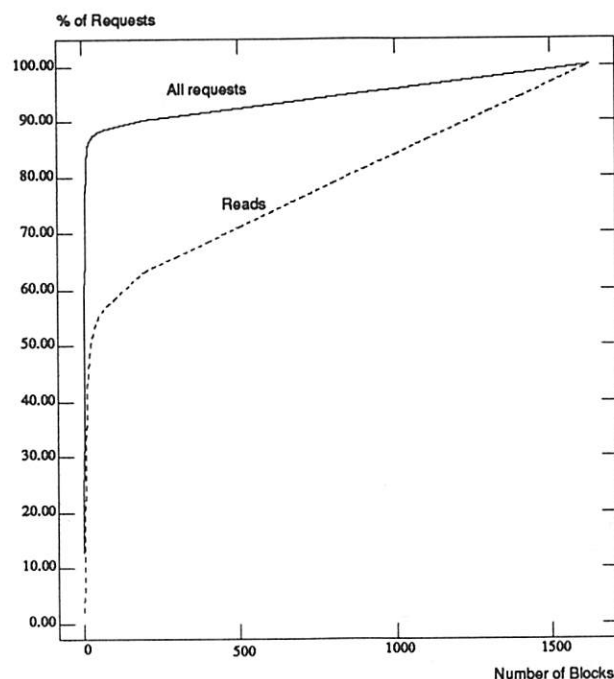


Figure 5: Distribution of block accesses on /usr.

Results for organ-pipe placement is included for comparison. Table 5 presents the results for another day for each of the interleaved and serial placement policies.

The serial placement policy projects hot blocks from different parts of the disk onto the reserved area without considering their access frequencies. It is a simpler policy than the organ-pipe placement but it is not as effective as the organ-pipe placement in reducing seek times. It fails to place more frequently accessed blocks closer to each other in the reserved area. As it can be seen in Tables 5 and 4 serial placement produces fewer zero-length seeks than organ-pipe does. The serial placement policy's relatively poor performance illustrates the importance of using block access frequencies to determine placement.

The performances of the interleaved placement policy and the organ-pipe placement policy are very close. Organ-pipe placement performs slightly better in terms of seek times and service times.

The interleaved placement policy is a variation of the organ-pipe policy. It tries to preserve the file system's rotational optimization in the reserved area. Organ-pipe placement may disturb the rotational optimization. While accessing files sequentially, this may result in higher rotational delays than in the case of no rearrangement.

Table 6 lists some data about the effects of each of the placement policies on rotational delays. These data are compiled from Tables 2-5. The values in Table 6 are the differences between the mean service times and the mean seek times for the read requests. They correspond to the sum of mean transfer times and mean rotational delays. Since rotational optimizations rarely benefit write requests, only the values for the read requests are included in the table. There are results from two different days for each the placement policies and for the no-rearrangement case.

Request size distributions are almost identical for each of the days, so we can safely assume that the differences in the values in Table 6 are due to differences in rotational delays. We can see from the values in the table that the organ-pipe placement and serial placement policies increase the rotational delays slightly. This is because of the disturbances they cause in the file system's rotationally-optimized placement. The interleaved placement, on the other hand, does not increase the rotational delays.

Interleaved placement, however, disturbs the organ-pipe arrangement of blocks in the reserved area.

	Organ-pipe		Interleaved		Serial	
	All Requests	Reads	All Requests	Reads	All Requests	Reads
FCFS Mean Seek Dist (cyls)	225	165	208	144	208	142
Mean Seek Distance (cyls)	8	23	15	24	22	39
Zero-length Seeks (%)	88	67	83	61	26	39
FCFS Mean Seek Time (ms)	21.46	16.14	20.02	14.39	20.02	14.23
Mean Seek Time (ms)	1.55	4.49	2.50	5.86	8.50	8.57
Mean Service Time (ms)	22.95	24.18	23.71	24.31	28.53	27.8
Mean Waiting Time (ms)	50.03	5.47	46.85	5.14	61.32	6.32

Table 4: Experiments with placement policies.

	Interleaved		Serial	
	All Requests	Reads	All Requests	Reads
FCFS Mean Seek Dist (cyls)	214	151	200	138
Mean Seek Distance (cyls)	12	35	23	39
Zero-length Seeks (%)	86	60	30	39
FCFS Mean Seek Time (ms)	20.59	14.97	19.57	13.91
Mean Seek Time (ms)	2.11	5.98	8.29	8.49
Mean Service Time (ms)	23.50	24.45	28.49	27.81
Mean Waiting Time (ms)	47.06	4.91	55.91	6.11

Table 5: Experiments with interleaved and serial placement policies.

As seen in Tables 5 and 4, this results in higher seek times than in the case of the organ-pipe placement. The increase in the mean rotational latency caused by the organ-pipe placement is more than made up for by the organ-pipe heuristic's higher seek times reductions. As a result, organ-pipe placement gives slightly better improvements in service time than interleaved placement.

4.2.4 Rearrangement on a Read/Write Mounted File System

In this subsection we will present the results of rearrangement experiments on a read/write mounted file system. As we mentioned earlier, the disk stored the home directories of 10 users. Because of the size limitations of the disk, we were not able to accommodate more users. This fact combined with the large buffer cache provided by SunOS resulted in very light disk loads.

For this environment we conducted experiments similar to the first set of experiments. We applied block rearrangement on alternate days for several weeks. In Figure 6, we present results from four consecutive days. Results for the other days are similar. In the figure mean seek distances and mean seek times are given. We show both arrival-order (FCFS) and actual seek delays. As usual statistics are given for read

	Mean Rotational Latency + Mean Transfer Time (ms)	
Without Rearrangement	18.49	18.72
Organ-pipe Placement	19.71	19.03
Serial Placement	19.32	19.26
Interleaved Placement	18.45	18.47

Table 6: Effects of different placement policies on rotational delays. The values in the table are mean rotational delay + mean transfer time for read requests in milliseconds.

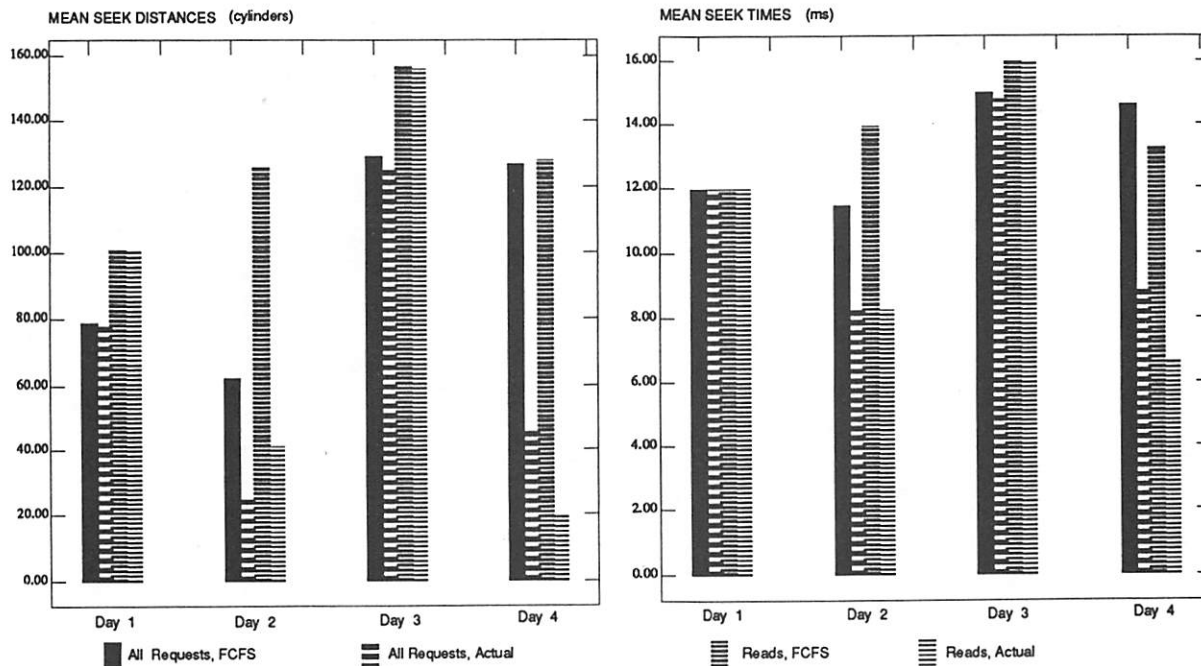


Figure 6: Results of experiments with read/write mounted file system.

requests as well as for all the requests. Block rearrangement was applied on days 2 and 4.

The reductions in the seek delays are not as large as those we observed for the `/usr` file system. Nonetheless, the reductions are significant, especially for the read operations. Mean seek times were reduced by 20%-40% for all the requests. Seek times for read requests were reduced by 40%-50%.

The poorer performance of the system in this environment can be attributed to several factors. Firstly, there are write requests resulting from new file creation and file expansion operations in this environment. Block rearrangement cannot benefit these kinds of write requests. Secondly, the original seek distances were relatively shorter. This makes the reductions look smaller. Finally, and most importantly, the block access distributions were less skewed in this environment. This can be observed from Figure 7. In this figure we have plotted the block access distribution for a typical day. Comparing Figure 7 with Figure 5, we see that the reference skew in this environment is less than that of the `/usr` file system.

A less skewed block access distribution is a disadvantage for the block rearrangement system. As the access distribution for the rearranged blocks becomes more uniform, the distribution of requests over the cylinders in the reserved area will also become more uniform. This reduces the system's ability to increase the number zero-length seeks.

Although it is not the case in this environment, a less skewed block access distribution may be a disadvantage in another way: the blocks placed in the reserved area get a smaller fraction of the total number of requests. As a result the disk head spends less time over the reserved area. This may reduce the benefits of the block rearrangement.

5 Conclusion

We have described the implementation of an adaptive block rearrangement technique under a UNIX operating system. The technique is implemented in the disk device driver and it is transparent to the rest of the system.

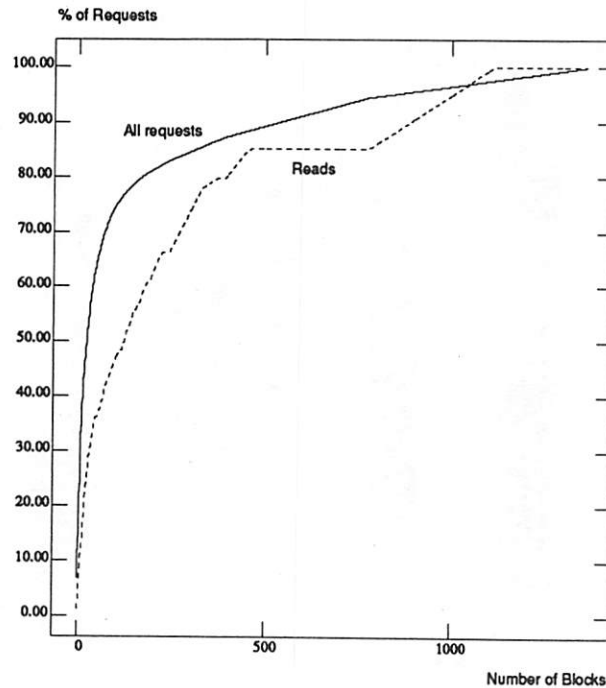


Figure 7: Distribution of block accesses on read/write mounted file system.

Measurements taken during the operation of the system on a network file server has shown that it is very effective in reducing disk service times by reducing seek times. Results also show that waiting times are reduced, especially for bursty arrivals. Disk space overhead of the block rearrangement technique is very low. Most of the benefits of block rearrangement are obtained by rearranging around 1% of all the blocks on a disk.

We have experimented with three placement policies for the rearranged blocks. The organ-pipe placement policy had the best performance in reducing seek times and service times. The interleaved placement policy's performance was close to the organ-pipe placement but slightly worse. the serial placement policy performed much worse than the other two.

The success of the block rearrangement technique is also closely tied to the skew in the block access distribution. The more skewed the distribution, the greater the improvement.

The experimental results shown in this paper were obtained using a 135 megabyte SCSI disk. We are planning to repeat the experiments on a larger disk in future.

Acknowledgements

We would like to thank Ólafur Gudmundsson, James da Silva, Harry Mantakos and the technical staff at the University of Maryland's Computer Science Department for their help and suggestions in setting up the experimental environment. We also would like to thank the people who agreed to be the users of our system and trusted our system with their files.

References

[Akyurek 93] Akyurek, Sedat, Kenneth Salem, "Adaptive Block Rearrangement," Proceedings of Ninth

International Conference on Data Engineering, Vienna, Austria, April 1993.

- [Carson] Carson, Scott D., Meenal Jobalia, "Precision Measurement of Disk Delay Parameters," in preparation.
- [English 92] English, Robert M., Alexander A. Stepanov, "Loge: A Self-Organizing Disk Controller," Proceedings of the Winter 1992 USENIX Conference, San Francisco, CA, 1992.
- [Floyd 89] Floyd, Richard A., Carla Schlatter Ellis, "Directory Reference Patterns in Hierarchical File Systems," IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, June 1989.
- [Ford 91] Ford, Daniel A., Stavros Christodoulakis, "Optimizing Random Retrievals from CLV format Optical Disks," Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, Spain, September, 1991.
- [Grossman 73] Grossman, David D., Harvey F. Silverman, "Placement of Records on a Secondary Storage Device to Minimize Access Time," JACM, Vol.20, No.3, July 1973.
- [McKusick 84] McKusick, K. Marshall, et al, "A Fast File System for UNIX," ACM Transactions on Computer Systems 2(3), August 1984.
- [McVoy 91] McVoy, L.W., S.R. Kleiman, "Extent-like Performance from a UNIX File System," USENIX Winter 1991 Conference Proceedings, Dallas, TX, 1991.
- [Ouster 85] Ousterhout, John K., et al, "A Trace Driven Analysis of the UNIX 4.2 BSD File System," Proceedings of the 10th ACM Symposium on Operating System Principles, 1985.
- [Rosenblum 91] Rosenblum, M., J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, Vol.10, February 1992, 26-52.
- [Ruemmler 91] Ruemmler, C., J. Wilkes, "Disk Shuffling", HPL-91-156, Hewlett-Packard Laboratories, Palo Alto, CA, October, 1991.
- [Salem 92] Salem, Kenneth, Daniel Barbará, Richard J. Lipton, "Probabilistic Diagnosis of Hot Spots," Proceedings of Eighth International Conference on Data Engineering, Tempe, Arizona, February 1992.
- [Staelin 91] Staelin, Carl, Hector Garcia-Molina, "Smart Filesystems," Proceedings of the Winter 1991 USENIX Conference, Dallas, TX, 1991.
- [Vongsath 90] Vongsathorn, Paul, Scott D. Carson, "A System for Adaptive Disk Rearrangement," Software-Practice and Experience, Vol. 20(3), March 1990.
- [Wong 80] Wong, C. K., "Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems," Computing Surveys, Vol.12, No.2, June 1980.

Author Biographies

Sedat Akyürek is a PhD candidate in the Department of Computer Science at the University of Maryland. His research interests include disk and IO systems, operating systems, databases and distributed systems. He is currently working on data replication techniques in disk systems. He has received an M.S. in Computer Science from the University of Maryland in 1991 and a B.S. in Computer Engineering from the Middle East Technical University, Turkey in 1988. Sedat Akyurek can be reached via email at akyurek@cs.umd.edu.

Kenneth Salem is an assistant professor in the Department of Computer Science at the University of Maryland, College Park, and a staff scientist at NASA's Center of Excellence in Space Data and Information Sciences, located at the Goddard Space Flight Center. His research interests include database and operating systems and transaction processing. Dr. Salem received a BS in electrical engineering and applied mathematics from Carnegie-Mellon University in 1983, and a PhD in computer science from Princeton University in 1989. He is a member of the ACM and the IEEE Computer Society. His email address is salem@cs.umd.edu

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- * sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- * fostering innovation and communicating both research and technological developments,
- * providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators' Guild

The System Administrators' Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- * Subscription to *;login:*, a bi-monthly newsletter;
- * Subscription to *Computing Systems*, a refereed technical quarterly;
- * Discounts on various UNIX and technical publications available for purchase;
- * Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- * The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- * The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

Frame Technology Corporation	Quality Micro Systems
Matsushita Electric Industrial Co., Ltd.	Sun Microsystems, Inc.
mt Xinu	Sybase, Inc.
Network Computing Devices, Inc.	UUNET Technologies, Inc.
Open Software Foundation	UNIX System Laboratories, Inc.

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-50-2